AUTOMATIC POLICY SATISFACTION AND VERIFICATION FOR INTRANET WIDE DEFENSE SYSTEMS USING SIGNATURE BASED INTRUSION DETECTION AND RESPONSE SYSTEMS

By

PETER MCKENZIE MELL
B.A. (Vanderbilt University) 1995
B.S. (Vanderbilt University) 1995
M.S. (University of California, Davis) 1998

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in Charge

1998

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: Introduction and Background

1.1 THE USE OF INTRUSION DETECTION AND RESPONSE SYSTEMS IN INTRANET SECURITY

The computer security industry is using intrusion detection and response systems (IDRSs) to defend intranets[1] alongside more traditional prevention mechanisms such as firewalls and authentication systems. We refer to any device, including a firewall, that performs either intrusion detection or intrusion response as an IDRS. Popular commercial IDRSs include Real Secure from ISS, Intruder Alert from Axent, Net Ranger from Wheelgroup, and Stalker from Network Associates [1] [2] [3] [4]. Advanced research products under development include Network Radar from Net Squared, EMERALD from SRI, and MIDS and GRIDS from the University of California at Davis [5] [6] [7] [8]. Many useful and free research products are available: tripwire, COPS, SPI-Net, DoSTracker, and NIDES [9] [10] [11] [12] [13].

Anomaly based IDRSs, like NIDES and EMERALD[2], detect anomalous activity on a host or in a network. Anomaly detectors have high false positive rates because anomaly engines report activity that deviates from the norm instead of reporting known attacks. Because of the high false positive rate, many do not connect anomaly detectors to response systems. In theory anomaly based systems may detect variations on known attacks and even unknown attacks, however, the high false positive rate hinders their widespread use.

Signature based IDRSs, like Net Ranger and MIDS, detect attacks by looking for activity that occurs during a particular attack. These systems typically use network sniffers and host audit logs as data sources. Single network interfaces or hosts almost always detect attack signatures because distributed signatures are difficult to write and have not yet been proven to be effective. Upon detecting an attack, the detection engines usually alert the firewall protecting the site[3] and it initiates a response. Possible responses include filtering packets or killing connections but can involve injecting arbitrary packets into the network, turning off Ethernet switches, or killing processes on hosts. Signature based IDRSs are the most popular today because of their relatively low false positive rate. Our work will apply only to signature based IDRSs.

The majority of signature based IDRSs protect at most a single site and thus have limited use in defending large intranets. The problem with using single site IDRSs is that sites can not work collectively to detect and respond to attacks. Single site IDRSs include Stalker, Network Radar, COPS, and tripwire. It is possible to install these products at many locations throughout a large intranet but the separate installations will not work together as a unified defense system.

Many signature based IDRSs solve this problem by creating a communication infrastructure that allows IDRSs throughout the intranet to communicate. A weakness of these systems is that none of them allow sites within an intranet to have intrusion detection and response policies. Real Secure, Net Ranger, GRIDS and MIDS provide no formal detection and response policies for sites. Intruder Alert enables one to divide an intranet into sites and then give security mechanisms in each

---

[1] We define an intranet, or autonomous system, as the networks and hosts that comprise a single administrative domain where all hosts have a path to all other hosts using only networks in the administrative domain. Furthermore, routers or firewalls separate the administrative domain from hosts outside the administrative domain.

[2] EMERALD and NIDES include some signature based detection as well as anomaly detectors.

[3] We define a site to have the same definition as an intranet except that sites are always subsets of some intranet.

site detection and response policies. However, Intruder Alert assigns detection and response policies to the security components instead of the sites.

The distinction between assigning a policy to security components within a site and the site itself is important. If a security component holds the detection and response policy, then the security component has a guaranteed level of security. If a site holds the detection and response policy, the site has a guaranteed level of security. Policies assigned to individual security components guarantee a site nothing because a site's security policy depends on the location of each security component and the interrelationships between their policies. Assume that one is attempting to protect a site that has a complex network topology with heterogeneous types of security components. It is difficult to give a site a well-defined level of protection by configuring a large number of components with various relationships. It is much easier to give the site a detection and response policy and then attempt to configure the security components to meet that policy. It is even better if we give a site a policy and then have the security components automatically placed within the site and automatically configured to meet the site's policy.

The detection and response policy at a site defines a level of protection needed for the site. A site can then compare the intranet configuration with its policy and prove that it has obtained the stated level of protection. Without requiring a well-defined level of protection, it is difficult to measure the security provided by a site, to discover what part of a site's security configuration is inadequate, or to prove that a site has any guaranteed level of protection.

1.2 THE PROBLEM

How can signature based IDRSs be used to defend an intranet such that every site has an automatically satisfied detection and response policy? In addition, how can the intranet defense system use a minimal number of IDRSs and yet allow each site to meet its policy? By "automatically satisfy" we mean that each site makes its own policy and then the intranet wide security mechanism configures itself to meet each site's policy.

Every site should have its own policy because every site has different security needs. The intranet wide security mechanism should automatically satisfy each site's policy because it is difficult to manually configure an intranet to meet hundreds of site's policies. In order to reduce the total cost of implementing the system, our problem requires any solution to minimize the number of IDRSs.

1.3 PREVIOUS WORK

The Boeing Intruder Detection and Isolation Protocol (IDIP) provides an intrusion detection and response framework [14]. The framework enables one to install IDRSs in an intranet to form an intranet defense system. It advances beyond commercial technology in that IDIP allows heterogeneous IDRS to work together in a plug and play manner.

The IDIP project demonstrates that it is possible to wrap IDRSs with completely different functionality and has them work seamlessly together using their attack description language. This capability is the framework needed in order to allow sites to have arbitrary intrusion detection and response policies. Most intrusion detection systems work only with a single product and this limits a site's detection and response policy to use only the functionality provided by that product. Using IDIP a site can decide upon the best detection and response policy and then buy components to help it meet that policy.

While IDIP provides the implementation to allow heterogeneous IDRSs to work together as an intranet defense system, it does not address policy. There exist sites in IDIP but each site has an ad hoc policy defined by the configuration of its security components.

The Naval Research Laboratory has published work that enables one to analyze an intranet to determine the percentage probability that a hacker can break into a particular site [15]. This work enables a site to use a detection and response policy stating that a hacker should have at most a specific percent chance of penetrating the site's defenses. Calculating the probability that a hacker can penetrate a site is difficult and so we have chosen to avoid using probabilistic policies. Instead, our policies require the detection of a set of attacks and require initiation of a set of responses upon an attack being detected. In the future our policies may include the percentage chance that an IDRS detects an attack at a specific network interface.

1.4 ASSUMPTIONS

We model IDRSs as collections of "security servers". Each security server defends an interface into a site or a set of hosts within a site. The security server initiates responses at arbitrary times as ordered by a wrapper over the IDRS. The wrapper also tells the security server which attacks to detect and for which sites.

IDRSs always detect attacks for which they have a signature. This assumption is justifiable because most common network attacks have simple signatures. In addition, dropped packets are not a concern because firewall and router based IDRSs do not drop packets.

We assume that the implementation of the distributed security system that we describe is correct. We prove that our design has certain characteristics but only with a correct implementation can we guarantee its functionality.

Our work covers only single path network attacks in an IP like environment. This means that for each attack there exists a single network path from the attacker to the target on which all attack packets travel. The path from the target back to the attacker may be different that the path from the attacker to the target. Thus, we do not address attacks where it is necessary to see information passing through multiple network paths to identify the attack.

We assume that all responses take place fast enough to be effective against each detected attack. While it is important to analyze what to do if a response occurs too late, it is beyond the scope of our work.

All messages sent between sites use public and private key cryptography to ensure authentication and integrity. This is important so that sites can not take down the security system by forging security configuration requests.

1.5 OVERVIEW OF OUR APPROACH

We solve the problem by designing a distributed security application that uses IDRSs as base components. We prove that our design has the properties that solve our problems. We allow each site to have its own detection and response policy and to control its own IDRSs. However, sites can communicate with each other in order to collaboratively defend themselves. The collaboration allows sites to not install all of the IDRSs that their policy might otherwise require. In this way our model can minimize the number of IDRSs used in an intranet and still allow each site an independent policy.

An important aspect of our model is that the defense system automatically configures IDRSs in an intranet to meet the detection and response policies of all sites in an intranet. The defense system requires no configuration of IDRSs and it allows addition and removal of IDRSs as the system functions. When a site removes an IDRS or the IDRS is not functioning, the defense system automatically configures the other IDRSs to attempt to make up for the loss. Sites may change their detection and response policies at any time and the defense system automatically reconfigures the IDRSs in the intranet to satisfy the new policy.

Since we allow sites to work together, we must have a concept of trust. Trust is a negative concept in that if site X trusts site Y then X is willing to not detect attacks coming in from Y. Figure 1.2.1 shows a common example of using trust. Site, X, trusts a site Y to provide detection and response services for itself. X then has the IDRS capability of Y since no site can reach X without the attack going through Y's IDRS. The difference between this scenario and the scenario where X installs all of its own IDRS systems is that Y is able to attack X without detection or reprisal. This requires us to carefully evaluate the effect of trust on the overall intranet security system.

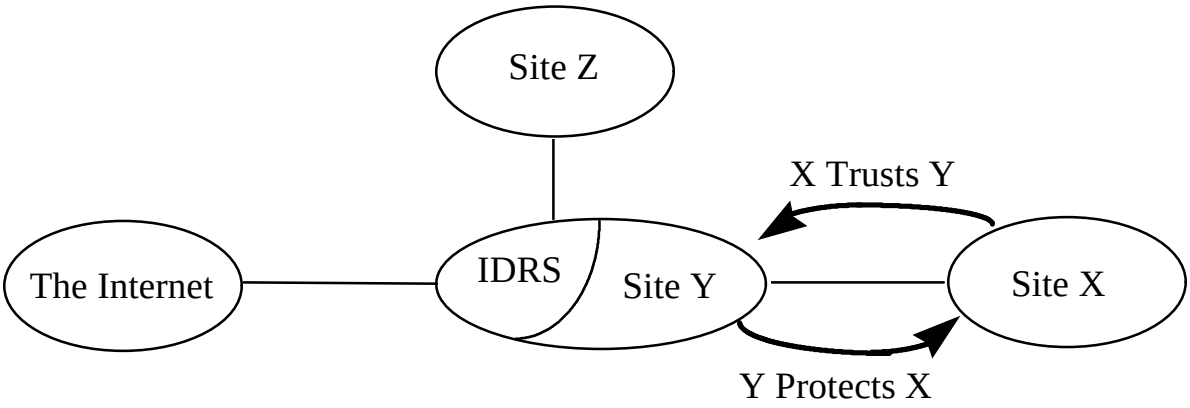


Figure 1.2.1 Example intranet used to demonstrate the sharing of IDRSs between sites

We explore automated techniques for minimizing the number of IDRSs needed in the network. We show that finding the minimal number of IDRSs possible is NP complete and so we give an approximation algorithm. In addition, we analyze several techniques that help a human optimize the IDRS placement.

Our design allows each site to have its own detection and response policies. The defense system automatically configures the IDRSs in the intranet to satisfy each site's policy. In addition, it provides mechanisms to minimize the number of IDRSs needed in an intranet to satisfy every site's policy. These two features solve our stated problem and hopefully this design will lead others to build economical intranet defense systems that give sites guaranteed levels of security and allow for the use of IDRSs from many vendors.

Chapter 2: The Security Model, its Algorithms, and Proofs

2.1 A DETECTION AND RESPONSE POLICY LANGUAGE

The primary goal of our work is to verify that the configuration of IDRSs in the intranet enable each site to meet its detection and response (DR) policy. While we want each site to have its own DR policy our work will be intractable if each site uses its own DR policy language. Thus we give a DR policy template or language that a site's security policy can use to create a specific DR policy.

We will always refer to a site's overall security policy as its "security policy" and the detection and response policy as its "DR policy". The "DR policy template" is the policy language we use to create DR policies.

Definition of trust: If site A trusts site B then both A is willing not to detect an attack against itself originating from B and A is willing not to be able to respond to an attack against itself originating from B.

DR Policy Template: A site, X, meets its DR policy iff it meets the following constraints:

1. X's security policy defines a set of pairs, ATT, where each pair consists of an attack and a response. Each attack in ATT must take place via network connections. X's security policy must also define a set of trusted site, TRUST.
2. If a site not trusted by X launches an attack listed in X's ATT set against X, then X or a site X trusts must detect the attack.
3. If a site not trusted by X launches an attack listed in X's ATT set against X, then X or a site X trusts must initiate a response on the attack path. The response taken must be the one paired with the attack in X's ATT set.

We introduce the concept of trust so that every site does not have to completely protect itself against every attack[4]. Trust allows sites to group together to create a common defense thereby allowing for a much cheaper implementation of our security model. The transitive property of trust does not cause any theoretical problems in the model but it can cause practical problems with certain implementations.

The DR policy template constraints assume that responses will occur at network interfaces between sites. Non-network responses are important but are beyond the scope of this model. One can incorporate non-network responses into an implementation of our model by setting the security policy of a site to take a non-network response upon detection of an attack.

The second and third DR policy template constraints force a site, X, to create a perimeter around itself for each attack and for each response in ATT as shown in figure 2.1.1. A detection and its related response do not have to have the same perimeters. Inside a perimeter exists only sites that X trusts and the DR policy requires that the detection or response take place on each perimeter.

[4] Remember that a site may be as small as a single host and thus may not have the resources to defend itself.

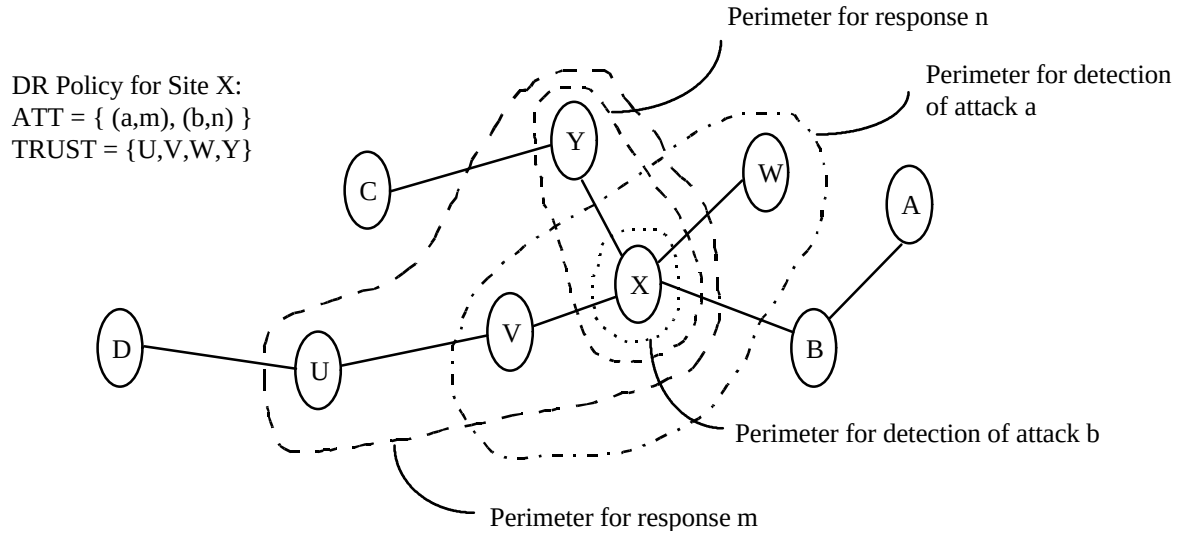


Figure 2.1.1 Example detection and response perimeters

*2.2 **EXAMPLES OF PRACTICAL USES FOR THE DR POLICY TEMPLATE***

This work is useless unless the DR policy template is useful in practice. To demonstrate that the DR policy is useful, we give examples of how DR policies solve current as well as important intranet defense problems.

Assume there is a company, Widgets Inc., with the simple intranet shown in figure 2.2.1. We want to ensure an attacker does not violate or shut down site X, Widgets' research site. Site Z has an interface to a leased line that connects Widgets' intranet with Widgets' corporate partner, Whipits Inc. Site Y has Widgets' Internet connection. Site W is where Widgets' trains their customers on the use of their product.

Widgets configures routers R1, R2, and R3 to drop all packets where both source and destination IP addresses exists in X, Y, or Z. This configuration stops some attacks and does not cut off any legitimate traffic. R1 and R3 allow no packets to reach W and R2 only allows packets from W to travel to sites X, Y, and Z. With this configuration, Widget hopes to restrict customers from transferring out the actual data they see at the training site.

Widget's currently does not perform any intrusion detection but the routers R1, R2, and R3 each have response capability. We will explore how to add intrusion detection capability to the intranet to effectively detect and respond to a few popular attacks.

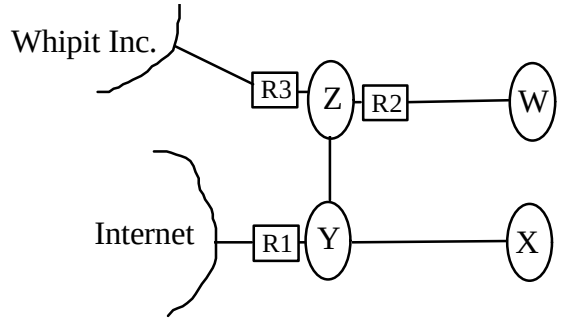


Figure 2.2.1 Example intranet to demonstrate intrusion detection and response

The DR policy is useful in protecting site X from the pepsi attack. See appendix A for details on the pepsi attack. Assume that only site X wishes to spend the resources needed to install an intrusion detection system. This means that X's detection perimeter encompasses only itself. X must trust Y and Z because no response devices exists between X and Y and also between Y and Z. X's response perimeter must travel through routers R1, R2, and R3 and encompass sites X, Y, and Z. Lastly, the response to a pepsi attack at routers R1, R2, and R3 can be to drop all packets whose destination is a character generation port and the source port is echo. This continues for 10 minutes.

Assume that a pepsi attack enters Widgets from either the Internet, W, or Whipit. The attack can not "connect" two hosts within X, Y, and Z because R1, R2, or R3 would filter out the attack packet. Thus, the attack packet must connect a host outside X, Y, and Z to a host in X. An IDRS at X, like the Network Radar, detects the attack and notifies site Y since X can not respond. Site Y can respond to the attack at R1 and so it does so. Y can not respond through its interface to Z and so it notifies Z of the attack. Z responds using R2 and R3. By dropping all packets to character generation ports for 10 minutes, the router can stop the current pepsi attack as well as any attacks within the next 10 minutes. The attacker can launch the attack every 10 minutes but the IDRSs will halt each attack immediately. Another possible response is to disable the echo or character generator services on applicable hosts.

While this defense works and does not hurt any legitimate traffic, why doesn't R1, R2, and R3 filter out pepsi packets all the time? Adding filtering rules to routers or firewalls can decrease their performance because they must apply the rules to every packet. Thus, it is often more cost effective to filter out an attack only when being attacked.

Now, assume that an attacker is launching the smurf attack against from somewhere in the Internet against a host in site X. See appendix A for a description of the smurf attack. X can detect the attack with an intrusion detection system such as Net Ranger. In response R1, R2, and R3 start to throttle the rate at which ICMP packets enter X for some small time period, say 10 minutes. X's network keeps working while causing minimal damage to our legitimate traffic. At the worst, we drop some but not all legitimate ICMP traffic from the outside entering X.

These last two cases showed the usefulness of intrusion detection and response in the case of denial of service attacks. Suppose now that a trainee at site W infiltrates site X and the intrusion detection system catches him entering a honey pot directory. See appendix A for a description of honey pots. Since X has no response capability it informs Y. Y also can not respond to this attack and so it informs Z. Z responds for X by killing the attacker's connection at R2. Z can then use R2 to prevent any new connections out of W until a human analyzes the means by which the hacker penetrated X.

2.3 THE MAPPING OF A SITE'S SECURITY POLICY TO A DR POLICY

Each site's security policy must use the DR policy template to create a DR policy. However, in our model the security policy is free to change the DR policy at any time. In addition to creating a DR policy, a site's security policy controls what services the IDRSs at the site may offer. For example, the site's policy could allow a firewall to kill connections when attacked but not to cut off all network traffic. Another example is that a site's policy might disallow an intrusion detection system to detect attacks on behalf of other sites.

The reason that a security policy is allow to change the DR policy at any time is to allow the DR policy to be more flexible. Examples are:

1. A site may wish to respond differently to attacks at night when employees generally are not working
2. A site may wish to temporarily disable some elements of ATT. This could happen when running network maintenance programs that would otherwise set off the IDRSs.

If an intranet configuration does not satisfy the new DR policy a site must negotiate a new intranet configuration that does satisfy the new DR policy, if possible. We assume that a site's security policy does not change the DR policy faster than the site can negotiate an intranet configuration that satisfies the new DR policy.

So far, we have introduced definitions. Now we will discuss the security framework in which a site can request an intranet configuration that satisfies its DR policy.

2.4 THE DUAL LEVEL FRAMEWORK

The model places security components on two levels. The higher level consists of components that hold a site's DR policy and negotiate with other sites to obtain an intranet configuration that satisfies their DR policy. The lower level consists of security servers that are IDRSs with some wrapper code that does the actual intrusion detection and response.

Figure 2.4.1 shows the high level view. Each site has two processes: the policy driver and the controller. The policy driver embodies a site's security policy, whatever that may be, and creates DR policies. The policy driver tells the controller when it wants to change the DR policy. The controller stores the current DR policy and is responsible for negotiating an intranet configuration that satisfies the site's DR policy. When controllers negotiate with other controllers, they only talk to physically neighboring sites in the network.

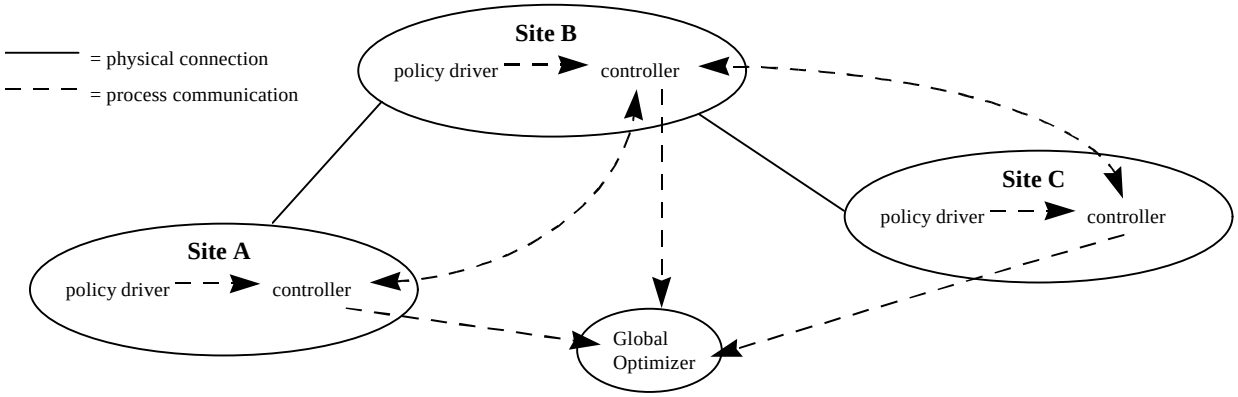


Figure 2.4.1 High level view of the security framework

The only other high level process is the global optimizer (GO). There exists only one GO for each intranet defense system. The GO gathers the state of every security component within the intranet in order to do global analysis.

Figure 2.4.2 shows a lower level view. Each site has a set of security servers that can perform services for its site and other sites. A service is either detecting a specific attack or initiating a specific response. Interface security servers monitor the entry points to the site, e.g.,

firewalls, routers, or sniffing IDSs. Internal security servers are host based security devices that protect the host on which they reside.

The policy driver and controller of a site communicate with that site's security servers. The security servers have no communication between themselves. The policy driver allows a security server to perform a set of services for certain sites. The controller requests that a security server perform a particular service for a site..

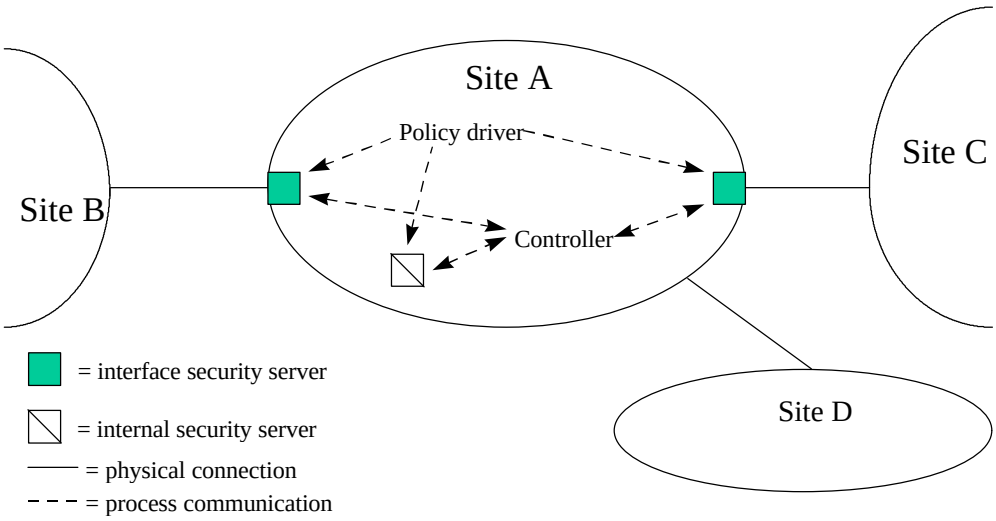


Figure 2.4.2 Low level view of the security framework

We shall show that this framework is sufficient to allow controllers to obtain and verify that an intranet configuration satisfies a site's DR policy. In addition, each site controls its own security components and yet the GO performs global analysis of the intranet. Each site has its own policy and yet all sites work together. Lastly, there does not exist a single point of failure[5].

2.5 DETAILS OF THE CONTROLLER

The controller at a site holds the site's DR policy and is responsible for negotiating with the other controllers to obtain an intranet configuration that satisfies the site's DR policy. The controller obtains the DR policy periodically from the policy driver. The relevant information is:

1. The set, ATT, of attack and response pairs
2. The set, TRUST, of trusted hosts

The controller communicates at the "low level" by talking with any security server in the controller's site. The controller may ask a security server for its state or it may change the security server's state.

[5] The GO is not a single point of failure as it has no role in the satisfying of site's DR policies.

Controllers communicate at the "high level" by talking with controllers in neighboring sites. This communication takes place using messages. Below we have defined a set of message types to enable a site to obtain a intranet configuration that satisfies its DR policy, if possible.

The parameters in the message types are:

1. X is the name of the site whose controller originally started the negotiation of which this message a part.
2. ∂ is a service type; that is, the name of either an attack or response.
3. TRUST is the TRUST set from X
4. id marks messages as belonging to a particular negotiation.

Assume for every message that a site Y sends each message through an interface ∝ to a site Z. The message types and their meanings are:

1. **service request (X, ∂, TRUST, id) :** Y makes a request to Z that it perform service ∂ for X through ∝. If ∂ is an attack, Y requests that Z detect any ∂ launched against X originating from a site not in TRUST, entering Z, and passing through ∝. If ∂ is a response, Y requests that Z promise to initiate ∂ against any attack against X that enters Z and passes through ∝. Z first attempts to provide ∂ at interface ∝. If this is not possible then Z attempts to provide ∂ at all of its other interfaces. For each interface on which it is not able to provide ∂ it will forward this **service request (X, ∂, TRUST, id)** message. We include TRUST in these messages because Y should never forward this message to a site Z which X does not trust. This message always originates at X and is only be forwarded by Y to Z.
2. **service granted (X, ∂, id) :** Y tells Z that for any attack entering Y and passing through ∝, Y promises to perform ∂. The **service granted (X, ∂, id)** message is sent as a positive reply to a **service request (X, ∂, TRUST, id)** message.
3. **service denied (X, ∂, id) :** Y tells Z that it is not performing service ∂ for X against any attack entering Y and passing through ∝. The **service denied (X, ∂, id)** message is a negative reply to a **service request (X, ∂, TRUST, id)** message.
4. **service overlap (X, ∂, id) :** Y tells Z that it previously received a **service request (X, ∂, TRUST, id)** message through another interface and that Y has already attempted to offer service ∂ through all of its interfaces. Controllers forward **service request (X, ∂, TRUST, id)** messages through the intranet in a tree structure with X as the root. When Z forwards a **service request (X, ∂, TRUST, id)** message and creates a cycle, Y sends a **service overlap (X, ∂, id)** message to alert Z of the situation. When Y receives a **service request (X, ∂, TRUST, id)** message through an interface, β, it attempts to provide service ∂ at β. If it can not provide ∂ at β then Y attempts to provide ∂ at or through all of its other interfaces. If Y then receives another **service request (X, ∂, TRUST, id)** message through ∝ it does not want to duplicate its work. Thus, it sends a **service overlap (X, ∂, id)** message saying that it has already processed the **service request (X, ∂, TRUST, id)** message.
5. **negotiation cancellation (X, id, TRUST) :** Y forwards a message originally from X saying that X wants to cancel all services requested using negotiation id. X does this only after successfully completing a new negotiation. X renegotiates an intranet configuration that meets its new DR policy whenever X's policy driver changes the DR policy or when a site breaks a promise it made to X.

6. **service cancellation (X, ∂, id) :** Y tells Z that for attacks entering Y and passing through ∝, Y is not performing service ∂. Z realizes that because of Y's message, it can not offer service ∂ for X and forwards the **service cancellation (X, ∂, id)** message through an appropriate interface. The controller that X trusts forward this message until it reaches X.
7. **service notification (X, ∂, id, data) :** If ∂ is an attack, Y tells Z that is detecting an attack ∂ against X. Data is the details of the attack. If ∂ is a response, Y is telling Z that ∂ needs to be initiated on the attack described in data. In addition, if ∂ is a response the data field contains a unique attack identification number that a site uses to uniquely identify response requests.

These messages are the primitives by which controllers communicate with their neighboring controllers. We present algorithms to use these primitives in the next section. These algorithms:

- Enable sites to negotiate intranet configurations in which their DR policy is satisfied
- Enable sites to cancel promises made in previous negotiations
- Enable sites to cancel whole negotiations

In order to use these distributed algorithms it is necessary for controllers to remember their state. Controllers store their state in three lists: SERVICING, ROUTING, and FULL_PROTECTED. Briefly:

1. SERVICING: Stores the services the controller at a site has promised to perform for itself and other controllers.
2. ROUTING: Serves as a routing table for all messages being sent to the originator of a negotiation. We do not use standard Internet routing protocols because messages must not get sent through a site that the originator of the negotiation does not trust.
3. FULL_PROTECTED: Stores the information about which **service request (X, ∂, TRUST, id)** messages the site is attempting to satisfy through all of its interfaces as opposed to satisfying only on the interface on which the message arrived.

The list, SERVICING, is a list of services the controller has promised to perform for other controllers. Each element of this list is a 4 tuple: (X, ∂, id, the interface on which ∂ was promised). **Service request (X, ∂, TRUST, id)** messages may cause a controller to add entries and **negotiation cancellation (X, id, TRUST)** messages and actions by each site's policy driver may remove entries. With this knowledge the controller at a site, Y, knows what services it has promised to perform, for whom it has promised these services, and through what interfaces. Y can use the id number in each entry to notify another site if it breaks a promise to perform a particular service.

The ROUTING list acts as a routing table for a site to send messages back to the originator of a negotiation. We do not use standard routing because it is important that messages pertaining to a particular negotiation pass only through sites which the originator of the negotiation trusts. Each negotiation in effect has its own routing tables of how packets should reach the originator of the negotiation. The ROUTING list contains 3 tuples: (X, id, the interface through which the site sends all messages to X). An entry will be added to the ROUTING list every time a controller receives a **service request (X, ∂, TRUST, id)** message and no entry with X and id exist in the ROUTING list.

The FULL_PROTECTED list records the services that the controller of the site has attempted to establish on all interfaces. A site attempts to satisfy a **service request (X, ∂, TRUST,**

id) message on the interface upon which the message arrives. If this is not possible a site tries to satisfy the **service request (X, ∂, TRUST, id)** message on all of its other interfaces. There is a need to distinguish if the site has attempted the "full" protection because the same protection request may arrive through multiple interfaces and the controller of a site needs to know whether or not it has previously tried to establish the service on all interfaces. The entries in the FULL_PROTECTED list will be 3 tuples: (X, ∂, id).

2.6 DETAILS OF THE SECURITY SERVERS

A security server is a wrapped IDRSs. The added software controls which attacks the IDRS detects and for whom, as well as which responses the IDRS initiates and for whom. Security servers come in two types: *interface* and *internal*. Interface security servers exist on or monitor the entry points to a site, and internal security servers exist on the hosts within a site's network. We model an IDRS that has both interface and internal components as multiple security servers.

Security servers communicate only with the policy driver and controller of the site in which the security server resides. Policy drivers and controllers use this communication to set the state of each security server within their site. Policy drivers and controllers thus "own" the security servers in their site.

The state of a security server must stay within the IDRSs capabilities. A security server can not detect an attack that the underlying IDRS can not detect. Security servers store their capabilities in the following set:

Capabilities of a security server, S:
1. The set of attacks S can detect
2. The set of responses S can initiate
3. The set of interfaces upon which S can offer its services

A security server defines its state by the following sets:

Configuration limitations for S (set by the policy driver):
1. The set of attacks S is allowed to detect for its site
2. The set of responses S is allowed to initiate for its site
3. The set of attacks S is allowed to detect for other sites
4. The set of responses S is allowed to initiate for other sites

Current Configuration for S (set by the controller that owns S):
1. The set of attacks S is detecting for its own site
2. The set of responses S is taking for its own site
3. The set of other sites for which S is detecting or responding
4. For each site in 3, the set of attacks S is detecting for this site
5. For each site in 3, the set of responses S is taking for this site

Interface security servers can detect and respond to attacks for sites other than the one where they reside because they monitor the interfaces. Internal security servers are host based and thus can only detect and respond to attacks at their own site. One should install internal security

servers on all relevant hosts within a site. We do not allow IDRSs that can not monitor any entry point into the site or can not protect every host within the site in this model.

2.7 THE NEGOTIATION ALGORITHM

This section presents the negotiation algorithm that allows the controller of a site to request an intranet configuration that satisfies a site's DR policy. In this section we give a brief overview of the algorithm, a detailed example, the algorithm itself, and proofs that the algorithm is correct.

Brief Overview of the Negotiation Algorithm

The negotiation algorithm is a distributed breadth first search algorithm where the controller at a site, X, negotiates with the controllers at other sites to obtain an intranet configuration where X meets its DR policy. The algorithm sets up a perimeter around X for each service required by the DR policy. Inside each perimeter exist only sites that X trusts. If a negotiation is successful, there will be no path from X to an untrusted site that does not go through some trusted site providing the services required by X's DR policy.

We generically refer to any detection or promises of future response that X needs as a *service*. For each service the controller at X tries to provide the service on each of its interfaces. If a controller can not protect an interface and X trusts the site on the other side of the interface, the controller sends out a message requesting protection on that interface. If X asks a site Y to provide service ∂ on an interface, $\propto$, then Y attempts to provide ∂ on $\propto$. If Y can not provide ∂ on $\propto$ then Y tries to provide ∂ on all of its other interfaces. If Y can not provide ∂ on one of its other interfaces, ξ, and X trusts the site on the other side of ξ, then Y forwards X's protection request through ξ and the site that receives the message follows the same algorithm as Y.

The action of trying to provide a service through some interfaces and sending out protection requests through other interfaces happens recursively. If a site attempts to forward X's protection request to a site X does not trust, the recursion terminates. Otherwise, the recursion stops when the service is provided on all paths from untrusted sites to X. If a controller attempts to forward a protection request to a site X does not trust then there exists a path from an untrusted site to X where no security server on the path provides the needed service. In this event X can not meet its DR policy.

Example 2.7.1: The Negotiation Algorithm

This example demonstrates the negotiation algorithm. Figure 2.7.1 shows a simple intranet setup and the security servers SS1..SS7 that are defending the network. Figure 2.7.2 shows the initial state of each of the seven security servers. Assume that the controllers have no entries in their data structures.

In this example we have the policy driver at A give its controller a DR policy. The controller at A uses the negotiation algorithm to create an intranet configuration that satisfies A's DR policy.

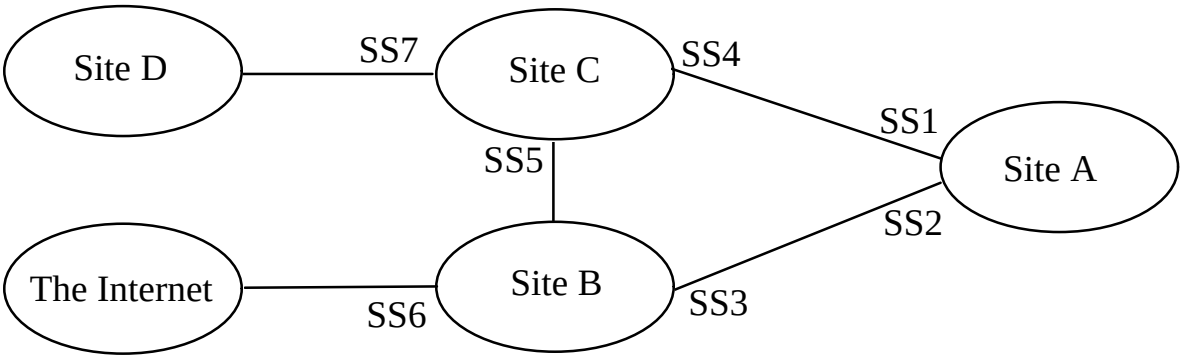


Figure 2.7.1 Example intranet used to demonstrate the algorithms

State of SS1 and SS2

Capabilities:
Can detect : {password guessing}
Can respond : {}

Policy Driver Limitations:
Allowed to detect : {all available}
Allowed to respond : {all available}
Allowed to detect for others : {all available}
Allowed to respond for others : {all available}

Current configuration:
Attacks detecting for A : {none}
Responses ready to initiate for A : {none}
Other sites being serviced : { }

State of SS3

Capabilities:
Can detect : {RBO}
Can respond : {}

Policy Driver Limitations:
Allowed to detect : {all available}
Allowed to respond : {all available}
Allowed to detect for others : {none}
Allowed to respond for others : {none}

Current configuration:
Attacks detecting for self : {RBO}
Responses ready to initiate for self : { none }
Other sites being serviced : { }

State of SS4 and SS5

Capabilities:
Can detect : {RBO}
Can respond : {connection killing}

Policy Driver Limitations:
Allowed to detect : {all available}
Allowed to respond : {all available}
Allowed to detect for others : {none}
Allowed to respond for others : {all available}

Current configuration:
Attacks detecting for self : {RBO}
Responses ready to initiate for self : {all available}
Other sites being serviced : { }

State of SS6, SS7

Capabilities:
Can detect : {RBO}
Can respond : {connection killing}

Policy Driver Limitations:
Allowed to detect : {all available}
Allowed to respond : {all available}
Allowed to detect for others : {all available}
Allowed to respond for others : {all available}

Current configuration:
Attacks detecting for self : {RBO}
Responses ready to initiate for self : {all available}
Other sites being serviced : {}

Figure 2.7.2 Initial state of the security servers in the example intranet

Assume that the policy driver at A gives the controller the following DR policy:
ATT = { (password guessing, kill connections from offending host to attacked port for 10 minutes),
(remote buffer overflow (RBO), kill connection that attempts to use the RBO) }
TRUST = { B, C }
See appendix A for a description of password guessing attacks and remote buffer overflow (RBO) attacks. Each element in the two pairs of the ATT set is a service that the controller must obtain and the controller puts these elements in a NEEDED_SERVICES list. The controller creates a unique identifier, say 34, to name the negotiation it is about to start. The controller attempts to obtain each service separately.

The controller queries its security servers and finds out that SS1 and SS2 are both able and allowed by the policy driver to detect password guessing attacks. The controller tells them to do so for site A. Since SS1 and SS2 cover all interfaces into A, the controller has successfully obtained the service. The controller adds two entries into its SERVICING list in order to remember why SS1 and SS2 are detecting the password guessing attack: (A, password guessing, 34, SS1 interface) and (A, password guessing, 34, SS2 interface).

The controller then attempts to obtain the next service listed in the NEEDED_SERVICES list, connection killing. Neither SS1 nor SS2 is able to provide connection killing and since X trusts the sites on the other side of the SS1 and SS2 interfaces, the controller sends a **service request (A, connection killing, {B,C}, 34)** out the SS1 and SS2 interfaces.

The controller at C receives the **service request (A, connection killing, {B,C}, 34)** message on the SS4 interface. Since SS4 is able and allowed to perform connection killing for A, the controller at C tells SS4 to prepare to perform this service for A. The controller at C adds an (A, connection killing, 34, SS1 interface) entry to its SERVICING list. It also adds an (A, 34, SS4 interface) to its ROUTING list in order to remember on which interface to send messages to A for this negotiation. The controller then sends a **service granted (A, connection killing, 34)** message through interface SS4.

Meanwhile, the controller at B receives a **service request (A, connection killing, {B,C}, 34)** message on its SS3 interface. Since SS3 is not able to provide connection killing the controller must try and provide "full protection". The controller can not provide connection killing on the interface where the request arrived and so it tries to provide connection killing on all of its other interfaces. The controller adds an (A, RBO, 34) entry to its FULL_SERVICING list and an (A, 34, SS3 interface) to its ROUTING list. Since SS6 is able and allowed by the policy driver to provide connection killing for A, the controller at B tells it to be ready to do so. The controller has no security servers on the B to C interface and since A trusts C it forwards the **service request (A, connection killing, {B,C}, 34)** message out of the B to C interface.

The controller at C receives the **service request (A, connection killing, {B,C}, 34)** message on its SS5 interface. The policy driver allows SS5 to provide connection killing and SS5 is able to provide connection killing. Thus, the controller tells SS5 to be ready to provide connection killing for A. The controller at C then adds an (A, connection killing, 34, SS1 interface) entry to its SERVICING list and an (A, 34, SS4 interface) to its ROUTING list. The controller at C then responds to B with a **service granted (A, connection killing, 34)** message through interface SS5.

The controller at B receives the **service granted (A, connection killing, 34)** message through its B to C interface and realizes that it has provided the "full protection". Thus it can send a **service granted (A, connection killing, 34)** message through interface SS3 to A.

The controller at A receives **service granted (A, connection killing, 34)** messages through both its interfaces and realizes that it has successfully obtained the connection killing service. The next service in the NEEDED_SERVICES list is RBO. Neither SS1 nor SS2 is able to provide detection of RBO and since A trusts both sites on the other sides of the SS1 and SS2 interfaces, the controller sends a **service request (A, RBO, {B,C}, 34)** out the SS1 and SS2 interfaces.

The controller at C receives the **service request (A, RBO, {B,C}, 34)** message through its SS4 interface and it attempts to provide detection of RBO against A for any attack entering C and going through the SS4 interface. The policy driver at C does not allow SS4 to detect RBO for A and so the controller at C adds an (A, RBO, 34) entry to its FULL_SERVICING list and tries to provide service RBO on all of its other interfaces. SS7 is able and allowed to detect RBO for A and so it detects RBO for A. The controller at C adds an (A, RBO, 34, SS7) entry to its SERVICING list to remember why SS7 is detecting RBO for A. The controller also adds a redundant (A, 34, SS4 interface) entry to its ROUTING list to remember how to send messages about negotiation 34 back to A. Since the policy driver at C does not allow SS5 to detect RBO for A, the controller at C must rely on B to detect RBO for A if an attack enters B and goes through the SS5 interface. Since the **service request (A, RBO, {B,C}, 34)** message says that A trusts B, C forwards the message to B through the SS5 interface.

Meanwhile, almost identical activity occurs at B as the controller at B receives the **service request (A, RBO, {B,C}, 34)** message through its SS3 interface. It attempts to detect RBO launched against A for any attack entering B and going through the SS3 interface. The controller at B does not allow SS3 to detect RBO for A and so it adds an (A, RBO, 34) entry to its FULL_PROTECTED list and tries to provide service RBO on all of its other interfaces. The policy driver at B allows SS6 to detect RBO for A and the security server at SS6 is able to detect RBO for A. Thus, the security server at SS6 detects RBO for A. The controller at B adds an (A, RBO, 34, SS6) entry to its SERVICING list to remember why SS6 is detecting RBO for A. The controller also adds a redundant (A, 34, SS3 interface) entry to its ROUTING list to remember how to send messages about negotiation 34 back to A. Since the controller at B has no security servers at the B to C interface, it must rely on C to detect RBO for A if the attack enters C and goes through the C to B interface. Since the **service request (A, RBO, {B,C}, 34)** message says that A trusts C, B forwards the message to C through the B to C interface.

The controller at C receives the **service request (A, RBO, {B,C}, 34)** message through the SS5 interface and realizes that it has a corresponding entry in its FULL_PROTECTED list. Therefore, the controller at C sends a **service overlap (A, RBO, 34)** message through the SS5 interface.

At the same time the controller at B receives a **service request (A, RBO, {B,C}, 34)** message through the B to C interface and realizes that it has a corresponding entry in its FULL_PROTECTED list. Therefore, the controller at B sends a **service overlap (A, RBO, 34)** message through the B to C interface.

The controller at B receives a **service overlap (A, RBO, 34)** message on its B to C interface and decides that it can rely on C to detect RBO for A on attacks going through that interface. Since SS6 is detecting RBO for A, the controller at B sends a **service granted (A, RBO, 34)** message through the SS3 interface.

At the same time, the controller at C receives a **service overlap (A, RBO, 34)** message on its SS5 interface and decides that it can rely on B to detect RBO for A on attacks going through that interface. Since SS7 is detecting RBO for A, the controller at C sends a **service granted (A, RBO, 34)** message through the SS4 interface.

A receives a **service granted (A, RBO, 34)** message through both of its interfaces and so it knows that no untrusted site can attack it with RBO without being detected. The last service on A's NEEDED_SERVICES list is connection killing but this was the first service obtained and so A does not need to do anything to obtain this service. The controller at A has negotiated an intranet configuration in which site A meets its DR policy. This negotiation set up the defense perimeters shown in Figure 2.7.3.

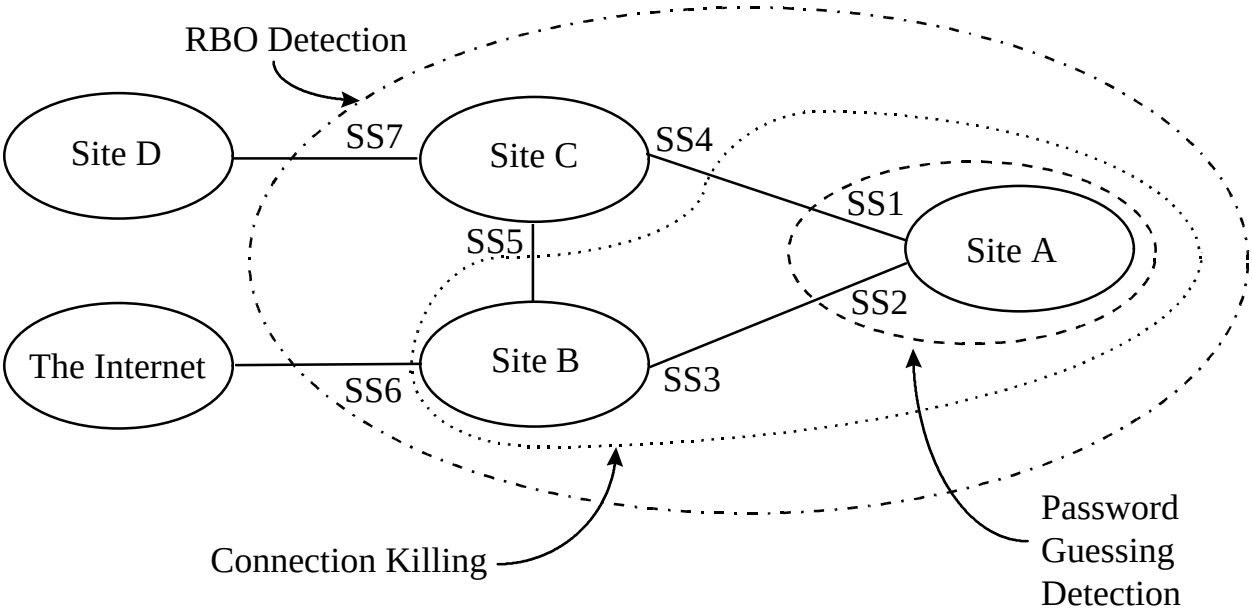


Figure 2.7.3 Defense perimeters created by example 2.7.1

Algorithm 2.7.1: The Negotiation Algorithm

The negotiation must be multithreaded to prevent deadlock. When a controller receives a message or starts a new negotiation it creates a thread to handle the task.

Algorithm 2.7.1.1 (for the controller at X to negotiate a secure intranet configuration):

1. Pick a unique id
2. Obtain the state of each security server X owns
3. Create a list, NEEDED_SERVICES, for each attack in ATT and each unique response in ATT.
4. For each "service", ∂, in NEEDED_SERVICES do:
    a. For each interface, $\propto$, where a security server is able and allowed to provide ∂, add (X, ∂, id, $\propto$) to X's SERVICING list. If ∂ is an attack begin detecting ∂ on that interface.
    b. For each interface, $\propto$, where no security server can provide ∂, if the site on the other side of $\propto$ is in X's TRUST set then send out a **service request (X, ∂, TRUST, id)** message, else exit the algorithm with X not secured.

c. For each interface out of which was sent a **service request (X, ∂, TRUST, id)** message, wait until the controller receives a reply.
d. If any of the replies are **service denied (X, ∂, id)** messages then exit the algorithm with X not secured

5. Exit the algorithm with X secure

Algorithm 2.7.1.2 (for a controller if it receives a **service request (X, ∂, TRUST, id)** message through an interface, ∝):

1. If there exists an entry (X, ∂, id) in FULL_PROTECTED then send a **service overlap (X, ∂, id)** message through ∝ and exit algorithm.
2. Add the entry (X, id, ∝) to the ROUTING list if the entry does not already exist.
3. Obtain the state of each security server at the site.
4. If a security server exists that is able and allowed to provide ∂ for X at interface ∝ then:
   a. Add (X, ∂, id, ∝) to the SERVICING list.
   b. If ∂ is an attack, tell the security server to begin detecting ∂ for X
   c. Send a **service granted (X, ∂, id)** message through ∝
   d. Exit the algorithm
5. Add the entry (X, ∂, id) to the FULL_PROTECTED list.
6. For each interface, except ∝, where a security server is able and allowed to provide ∂ for X, add (X, ∂, id, the interface) into the SERVICING list. If ∂ is an attack, tell the security server to begin detecting ∂ on that interface.
7. For each interface, except ∝, where no security server can provide ∂ for X, if the site on the other side of the interface is in the message's TRUST set then send out the **service request (X, ∂, TRUST, id)** message, else send a **service denied (X, ∂, id)** message through ∝ and exit the algorithm.
8. For each interface out of which was sent a **service request (X, ∂, TRUST, id)** message, wait until the controller receives a reply.
9. If any of the replies are **service denied (X, ∂, id)** messages then send a **service denied (X, ∂, id)** message out of ∝, else send a **service granted (X, ∂, id)** message out of ∝.

We now prove that the negotiation algorithm is correct. By correct we mean that it will declare that an intranet configuration satisfies a site's DR policy iff the intranet configuration satisfies the site's DR policy. Also, we prove that the negotiation algorithm will find an intranet configuration that satisfies a site's DR policy if such a configuration exists.

The following two lemmas prove that if the negotiation algorithm declares that a site meets its DR policy, that constraints 2 and 3 in the DR policy hold. Lemma 2.7.1.1 assumes that there exists a path on which security servers can not provide a requested service and it proves that each site on the path's FULL_PROTECTED list will list the service. Lemma 2.7.1.2 uses Lemma 2.7.1.1 to prove that if the negotiation algorithm processed a service, ∂, then DR policy constraints 2 and 3 hold for that service.

Lemma 2.7.1.1 : Assume that in an intranet there exists a path P with n+1 nodes, n>0, that consists of the sites X,S1,S2,…,Sn. Assume that service ∂ is not provided on P and that X trusts S1…Sn. If

the negotiation algorithm is run by X and it processes ∂, then each site S1…Sn has a (X,∂,id) entry in their FULL_PROTECTED lists.
Proof: (by induction on n)
Basis, n = 1:

If n=1 then P consists of X and S1. In order to process ∂ in the negotiation algorithm, the controller at X first tries to provide ∂ at X's interface to S1, β. The controller at X is not be able to provide ∂ at β since we assume that ∂ is not provided on P. The controller at X then sends S1 a **service request (X, ∂, TRUST, id)** message through β. S1 may already have a (X,∂,id) entry in its FULL_PROTECTED list due to its receiving a **service request (X, ∂, TRUST, id)** message on another interface and the basis holds. If S1 does not have a (X,∂,id) entry in its FULL_PROTECTED list then S1 receives the **service request (X, ∂, TRUST, id)** message on an interface ∝. The controller at S1 attempts to provide ∂ at ∝ but is not be able to do so because we assume that ∂ is not provided on P. Then the negotiation algorithm has S1 put a (X,∂,id) entry in its FULL_PROTECTED list.
Induction Hypothesis: S1..Sn-1 has a (X,∂,id) entry in its FULL_PROTECTED list.
Proof for n:

Sn-1 has a (X,∂,id) entry in its FULL_PROTECTED list by the induction hypothesis. The controller at Sn-1 can only put the entry in its FULL_PROTECTED list upon receipt of a **service request (X, ∂, TRUST, id)** message on some interface, ∝.

If ∝ is the interface at Sn-1 leading to Sn then Sn must have (X,∂,id) in its FULL_PROTECTED list. The reason is that by the negotiation algorithm, controllers only forward **service request (X, ∂, TRUST, id)** messages if they have a (X,∂,id) entry in their FULL_PROTECTED lists.

If ∝ is not the interface at Sn-1 leading to Sn then after adding the entry Sn-1 tries to provide ∂ on all of its interfaces except ∝. Let β represent Sn-1's interface to Sn. The controller at Sn-1 must not be able to provide ∂ at β because we assume that ∂ is not provided anywhere on P. Since the controller at Sn-1 is unable to provide ∂ at β, by the negotiation algorithm it forwards the **service request (X, ∂, TRUST, id)** message through β to Sn. The controller at Sn receives the **service request (X, ∂, TRUST, id)** message and checks its FULL_PROTECTED list for a (X,∂, id) entry. If it has such an entry then the proof holds. Otherwise, the controller at Sn tries and fails to provide ∂ at its interface to Sn-1 because we assume that ∂ is not provided on P. At this point the negotiation algorithm has the controller at Sn put a (X,∂,id) entry in its FULL_PROTECTED list.

Lemma 2.7.1.2 : If the negotiation algorithm declares that a site, X, meets its DR policy then for each service, ∂, processed by the negotiation algorithm, ∂ will be provided on all paths from sites X does not trust to X.
Proof: (by contradiction)

Assume that at least one path exists from a site X does not trust to X on which ∂ will not be provided. Let P be the shortest such path. If more than one is equally short then randomly pick one. Let Z be the site that X does not trust. Let Y be the site on P adjacent to Z. Except for Z, P consists of sites X trusts or else P would not be the shortest path.

By lemma 2.7.1.1 the controller at Y has a (X,∂,id) entry in its FULL_PROTECTED list. Controllers only put entries in their FULL_PROTECTED lists when they receive **service request (X, ∂, TRUST, id)** messages and so Y received at least 1 **service request (X, ∂, TRUST, id)** message. Let ∝ be the interface on which Y received the **service request (X, ∂, TRUST, id)** message that made Y put the entry in its FULL_PROTECTED list.

∝ can not be Y's interface to Z because there is no way for Z to get a copy of the **service request (X, ∂, TRUST, id)** message. Controllers only forward **service request (X, ∂, TRUST, id)** messages to sites in a message's TRUST set and X does not trust Z. Z can not forge a **service request (X, ∂, TRUST, id)** message because we assume that controllers use encryption on all messages to assure authenticity and integrity.

When the controller at Y put the (X,∂,id) entry in its FULL_PROTECTED list, the negotiation algorithm had the controller at Y try to provide ∂ on all interfaces except ∝. Thus, the controller at Y attempted to provide ∂ on its interface to Z. Since we assume that ∂ is not provided on P this attempt must have failed. Upon failure, the negotiation algorithm has the controller at Y attempt to forward the **service request (X, ∂, TRUST, id)** message to Z. However, since Z is not in the message's TRUST set, the negotiation algorithm has the controller at Y reply to the original **service request (X, ∂, TRUST, id)** message by sending a **service denied (X, ∂, id)** message out ∝.

By lemma 2.7.1.3 X will receive the **service denied (X, ∂, id)** message. When the controller at X receives a **service denied (X, ∂, id)** message, the negotiation algorithm does not declare that X meets its DR policy. Thus we have a contradiction and ∂ must be provided somewhere on P.

Lemma 2.7.1.3 : If a controller sends a message to a site X then the message will reach X.
Proof:

A site, Y, will only send a message to X if it is involved in some negotiation started by X. Y can only be involved in a negotiation started by X if it receives a **service request (X, ∂, TRUST, id)** message. Upon receipt of a **service request (X, ∂, TRUST, id)** message a controller adds a (X, id, ∝) entry to its ROUTING list. Once a controller has an entry with X and id in its ROUTING list it will add no more entries by step 2 of algorithm 2.7.1.2. Thus, each site has a most one entry in its ROUTING list that has X and id.

Make a graph, G, of n+1 nodes where the first n nodes correspond to sites that received X's **service request (X, ∂, TRUST, id)** message. Let the n+1 node represent X. Each of the first n sites has a single entry in its ROUTING list with X and id. Create an edge in G for each entry. Since sites only can forward a **service request (X, ∂, TRUST, id)** message if they first receive a **service request (X, ∂, TRUST, id)** message, we know that each site's ROUTING list entry will create an edge between two nodes in G.

There are n+1 nodes and n edges since X does not have any entries with X and id in its ROUTING list. By definition, G is a directed tree. Since G is a tree it has no cycles. Also, since X has no entries in its ROUTING list then the node corresponding to X, the n+1 node, must be the root. Thus, any messages leaving any of the first n nodes will be forwarded up to the n+1 node. This means that any message sent by a site to X will reach X.

Theorem 2.7.1 : If the negotiation algorithm declares that an intranet configuration satisfies a site's DR policy then the intranet configuration satisfies the site's DR policy.
Proof:

The DR policy template states that if a site, X, meets three constraints then X meets its DR policy. Assume that X meets constraint 1 as this is a prerequisite to using the negotiation algorithm. Now we need to show that if the negotiation algorithm declares that X meets its DR policy then X meets constraint 2 and 3.

The second security constraint says that for each attack, ∂, in X's ATT set, ∂ will be detected on any path from a site X does not trust to X. For each attack, ∂, in X's ATT set, the negotiation algorithm puts ∂ in its NEEDED_SERVICES in step 3. The negotiation algorithm processes each element of the NEEDED_SERVICES list in step 4. This combined with the fact that the negotiation algorithm did declare X secure means that by Lemma 2.7.1.2, for any attack path from an a site X does not trust to X, service ∂ will be provided on some interface on the path. That means for any attack path from a site X does not trust to X, attack ∂ will be detected along that path.

The third security constraint says that for any network path from a site X does not trust to X, a security server exists on the path that can initiate each response listed in ATT. For each response, ∂, in X's ATT set, the negotiation algorithm puts ∂ in its NEEDED_SERVICES in step 3. The negotiation algorithm processes each element of the NEEDED_SERVICES list in step 4. This combined with the fact that the negotiation algorithm did declare X secure means that by Lemma 2.7.1.2, for any attack path from an a site X does not trust to X, response ∂ will be provided on some interface on the path. That means that for any attack path from a site X does not trust to X, response ∂ can be initiated along that path.

Thus, if the negotiation algorithm declares that an intranet configuration satisfies X's DR policy then the intranet configuration does satisfy X's DR policy.

Now we prove the converse of theorem 2.7.1 by proving that if the negotiation algorithm does not find an intranet configuration in which a site meets its DR policy then no such configuration exists. Lemma 2.7.2.1 proves that as a **service denied (X, ∂, id)** message travels through an intranet, every site on that path can not provide a particular service for X. Lemma 2.7.2.2 proves that the site that originated the **service denied (X, ∂, id)** message is neighbors with a site that X does not trust and that the site can not provide ∂ at the right location to protect X. Theorem 2.7.1 uses these two lemmas to prove that if a site, X, receives a **service denied (X, ∂, id)** message then there exists no intranet configuration in which X can meet its DR policy.

Lemma 2.7.2.1: If a **service denied (X, ∂, id)** message enters a site S on an interface ∝ then S can not provide ∂ on ∝. In addition, S will forward the **service denied (X, ∂, id)** message out of an interface, β, and S will not be able to provide ∂ at β.
Proof:

By negotiation algorithm 2.7.1.2 steps 7,8, and 9; S can only receive a **service denied (X, ∂, id)** message on ∝ if it previously sent a **service request (X, ∂, TRUST, id)** message through ∝. S can only send a **service request (X, ∂, TRUST, id)** message through ∝ if it is attempting "full_protection" for ∂ and if it is unable to provide ∂ for X on ∝. Thus, S must have a (X,∂,id)

entry in its FULL_PROTECTED list. S can only have an (X,∂,id) entry in its FULL_PROTECTED list if it received a **service request (X, ∂, TRUST, id)** message through an interface β and if it was unable to provide ∂ on β.

After S sent the **service request (X, ∂, TRUST, id)** message through ∝, it blocked waiting for a reply. Upon receiving the **service denied (X, ∂, id)** reply on ∝, the negotiation algorithm forwards the **service denied (X, ∂, id)** message out of the interface upon which the **service request (X, ∂, TRUST, id)** message arrived that caused S to attempt the "full_protection". Thus, the **service denied (X, ∂, id)** message entered S through ∝ and was forwarded through β.

Therefore, ∂ could not be provided on ∝ and β by S and the **service denied (X, ∂, id)** message entered S at ∝ and left through β.

Lemma 2.7.2.2: If a site, W, originates a **service denied (X, ∂, id)** message and sends it through an interface, ∝, then W can not provide ∂ on ∝ for X. In addition, there exists an interface β at W such that W can not provide ∂ at β and the site on the other side of β is not trusted by X.
Proof:

W can only originate a **service denied (X, ∂, id)** message in step 7 of algorithm 2.7.1.2 and so W must be using algorithm 2.7.1.2. Since W reached step 7 we can infer several things about W. In order to be using algorithm 2.7.1.2 W must have received a **service request (X, ∂, TRUST, id)** message on some interface, ∝. In step 1 of the algorithm there must not have been a (X,∂,id) entry in W's FULL_PROTECTED list or else W would have exited the algorithm and never reached step 7. Also, W must not have been able to provide ∂ on ∝ for X or else the algorithm would have exited before step 7. In step 7 no action takes place unless there exists an interface, other than ∝, where W can not provide ∂ for X. Since W originates a **service denied (X, ∂, id)** message in step 7, we know three things:

1. The **service denied (X, ∂, id)** message is sent through ∝
2. There must exist an interface β, not equal to ∝, on which W can not provide ∂ for X.
3. The site on the other side of β must not be in the original **service request (X, ∂, TRUST, id)** message's TRUST set.

Therefore, W can not provide ∂ on ∝ for X, there exists an interface β at W on which W can not provide ∂ for X, and the site on the other side of β is not trusted by X.

Theorem 2.7.2 : If the negotiation algorithm does not find an intranet configuration which satisfies a site's DR policy then there exists no intranet configuration in which the site will meet its DR policy.
Proof: (by contradiction)

Assume then that the negotiation algorithm does not find an intranet configuration where a site, X, meets its DR policy but that there does exist an intranet configuration in which X meets its DR policy.

If the negotiation algorithm did not find an intranet configuration that satisfies X's DR policy then according to the negotiation algorithm, one of two events must have happened:

1. There exists a neighbor of X which is not trusted by X on an interface where the security servers at X can not provide one of the services, ∂, in the negotiation algorithm's NEEDED_SERVICES list.

2. X received a **service denied (X, ∂, id)** message

Assume that the intranet configuration does not satisfy X's DR policy because of event 1. If ∂ was an attack then X violates constraint 2 of the DR policy. If ∂ was a response then that violates constraint 3 of X's DR policy. Thus, there does not exist an intranet configuration where X can satisfy its DR policy.

Assume that the negotiation algorithm finds no intranet configuration that satisfies X's DR policy because of event 2. That means that X receives a **service denied (X, ∂, id)** message through an interface ∝. By step 4b-c of algorithm 2.7.1.1, X can only receive **service denied (X, ∂, id)** messages through interface upon which it sent a **service request (X, ∂, TRUST, id)** message. By step 4b, X will only send **service request (X, ∂, TRUST, id)** messages out of interface upon which it can not provide service ∂ for itself.

Since X received a **service denied (X, ∂, id)** message, that message must have originated at some site W. The message must have traveled along a path of sites W,S1..Sn,X for the message to reach X. By lemma 2.7.2.2 W has an interface β on which it can not provide ∂ for X and the site on the other side of β is not trusted by X. Also by lemma 2.7.2.2 W sends the **service denied (X, ∂, id)** message out of an interface ∝ on which it can not provide ∂ for X. By lemma 2.7.2.1 each site S1..Sn can not provide ∂ for X on the interfaces upon which the **service denied (X, ∂, id)** message entered and left the site. Thus, there exists a path from a site X does not trust to X on which ∂ can not be provided. If ∂ is an attack then we have a violation of constraint 2 of X's DR policy. If ∂ is a response then we have a violation of constraint 3 of X's DR policy. Thus we have a contradiction and there must not exist an intranet configuration in which X meets its DR policy.

Theorem 2.7.3 : The negotiation algorithm will not deadlock

Proof: (by contradiction)

Since controllers are multithreaded we will prove that no thread in a controller can deadlock. Assume that the negotiation algorithm will deadlock. By the negotiation algorithm, a controller at a site Y only blocks when it has put a (X,∂,id) entry in its FULL_PROTECTED list and is waiting for a reply from a **service request (X, ∂, TRUST, id)** message that it forwarded to a site Z.

If a controller has a (X,∂,id) entry in its FULL_PROTECTED lists then it immediately answers all incoming **service request (X, ∂, TRUST, id)** messages with a **service overlap (X, ∂, TRUST, id)** message. Remember that the negotiation algorithm specifies that controllers are multithreaded. Thus, a controller can respond to an incoming message while blocking waiting for a reply.

Now, for a site S1 to be deadlocked it must be waiting for a reply from a site S2. For S2 to be deadlocked it must be waiting for a reply from some site S3 and so on. Assuming there are n sites in the intranet, Sn can only block if it has sent out a **service request (X, ∂, TRUST, id)** message and is waiting for a reply. However, all sites S1..Sn-1 have a (X,∂,id) entry in their FULL_PROTECTED lists. Thus, Sn will get a reply in a finite amount of time. Sn then will respond to Sn-1 and so on until S1 has received a reply. Thus, it is impossible for S1 to deadlock.

2.8 MISCELLANEOUS ALGORITHMS

We now present several algorithms that allow sites to cancel promised services, cancel negotiations, tell other sites about attacks, and ask other sites to respond to an attack. We first present each algorithm and then argue that it is correct.

Algorithm 2.8.1: The Service Cancellation Algorithm

The service cancellation algorithm allows the controller at a site, Y, to cancel a service previously promised to a site, X. Using the algorithm, Y can notify X that it canceled a service it previously promised. Controllers cancel promised services because the policy driver at Y no longer allows Y to perform the service or because Y is no longer able to perform the service.

Algorithm to notify other sites of canceled promises

1. For each service, ∂, that the security server will stop providing do step 2.
2. For each interface, $\propto$, at which the security server will stop providing ∂ do step 3.
3. For each entry, (a,∂,c,$\propto$) in the SERVICING list do:
    a. If a is the name of the local site, inform the controller that the intranet configuration no longer satisfies its DR policy.
    b. If a is not the name of the local site, send a **service cancellation (a, ∂, c)** message out of the interface, β, specified in the ROUTING entry that matches a and c.
    c. Remove the entry from the SERVICING list

Algorithm for a controller if it receives a **service cancellation (X, ∂, id)** message

1. Search the ROUTING list to find an entry with X and id.
2. Forward the **service cancellation (X, ∂, id)** message out of the interface specified in the ROUTING list

We now argue that this algorithm is correct. Every time a controller at a site, Y, promises to perform a service for another site, X, the controller makes an entry in its SERVICING list. It also makes an entry in its ROUTING list so it knows out which interface to send messages to X pertaining to this negotiation. When Y cancels the promised service it sends a **service cancellation (a, ∂, c)** message through the interface specified in the ROUTING list. Sites trusted by X forward the message until the message reaches X. X then knows that Y canceled the service and that the intranet configuration no longer satisfies X's DR policy.

Algorithm 2.8.2 : The Negotiation Cancellation Algorithm

The negotiation cancellation algorithm enables a site, X, to cancel a previous negotiation that it established using the negotiation algorithm. We will refer to the previous negotiation's id as "old_id" and the previous TRUST set as "OLD_TRUST".

Algorithm for a controller at X to cancel a previous negotiation where id=old_id and TRUST= OLD_TRUST.

1. Search the SERVICING list and remove any entries that have both X and old_id.
2. Search the ROUTING list and remove any entries that have both X and old_id.
3. Search the FULL_PROTECTED list and remove any entries that have both X and id.
4. Get the state of all security servers at X

5. Compare the SERVICING list with the state of each security server and if any security server does not need to perform any service that it is performing, tell the security server to stop performing the service.
6. For each interface into X where the site on the other side is in the OLD_TRUST set, send a **negotiation cancellation (X, old_id, OLD_TRUST)** message.

Algorithm for a controller that receives a **negotiation cancellation (X, id, TRUST)** message on interface $\propto$

1. If the ROUTING list does not have any entries with id, exit the algorithm.
2. Search the SERVICING list and remove any entries that have both X and id.
3. Search the ROUTING list and remove any entries that have both X and id.
4. Search the FULL_PROTECTED list and remove any entries that have both X and id.
5. Get the state of all security servers at X
6. Compare the SERVICING list with the state of each security server and if any security server does not need to perform any service that it is performing, tell the security server to stop performing the service.
7. For each interface, except $\propto$, where the site on the other side is in the message's TRUST set, send a **negotiation cancellation (X, id, TRUST)** message.

In order to argue that this algorithm is correct we need to argue that controllers will forward X's **negotiation cancellation (X, id, TRUST)** message to all sites that have knowledge of the negotiation id and that each of these sites cleans up all records of the old negotiation. The **negotiation cancellation (X, id, TRUST)** message reaches all sites that know about the negotiation because each controller forwards the message out all interfaces that lead to sites that X trusts. Thus, all sites that X's initial run of the negotiation algorithm reached receive a **negotiation cancellation (X, id, TRUST)** message. When a site receives the **negotiation cancellation (X, id, TRUST)** message it cleans up its SERVICING, ROUTING, and FULL_PROTECTED lists. Then the controller makes sure that none of its security servers are providing any services that they no longer need to provide.

Since a site deletes all id entries from its ROUTING list when it receives a **negotiation cancellation (X, id, TRUST)** message, it can know if it receives the same message twice. Since the algorithm throws out **negotiation cancellation (X, id, TRUST)** messages if a site does not have id in its ROUTING list, controllers do not infinitely forward **negotiation cancellation (X, id, TRUST)** messages through the intranet.

Algorithm 2.8.3: The Attack Detection Algorithm

Our security model uses the attack detection algorithm to report attacks when the controller at a site, Y, detects an attack for a site X. The algorithm describes how Y notifies X of the attack.

Algorithm for a controller if it detects an attack, ∂, against X

1. Look through the SERVICES list for an entry with X and ∂.
2. Using the id from the entry in the SERVICES list, look through the ROUTING list for an entry with X and id.
3. Send a **service notification (X, ∂, id, data)** message out of the interface specified in the ROUTING list where data is the details of the attack.

Algorithm for a controller if it receives a **service notification (X, ∂, id, data)** message through interface ∝

1. If ∂ is an attack:
    a. Search the ROUTING list to find an entry with X and id.
    b. Forward the **service notification (X, ∂, id, data)** message out of the interface specified in the ROUTING list.
2. If ∂ is a response:
    a. The data portion of the message contains an attack identification number. If the controller has seen this identification number from site X for negotiation id, then exit the algorithm.
    b. Get the state of all security servers
    c. From each entry in the SERVICES list with X, ∂, and id as parameters, create a list of interfaces upon which ∂ needs to be initiated.
    d. For each interface β from step b, there will exist a security server that can enact response ∂ at interface β. Tell this security server to start initiating ∂ at β.
    e. For each interface, except ∝, upon which a response is not initiated, forward the **service notification (X, ∂, id, data)** message.

Assume that Y detects an attack against X and it promised to detect this attack for X. The controller at Y sends a **service notification (X, ∂, id, data)** message out of the interface specified in its ROUTING list. Sites X trust forward this message until it reaches X. The message informs X about the attack.

Algorithm 2.8.4 : The Response Initiation Algorithm

The response initiation algorithm implements a distributed response to an attack upon X that is, assuming the intranet configuration satisfies X's DR policy, guaranteed[6] to occur in the attack path. When the controller at a site, X, finds out that it is being attacked, it looks at its ATT set to determine the correct response. Assume the intranet configuration satisfies X's DR policy.

Algorithm for a controller at X if it finds out that it is being attacked by an attack, β

1. Let the_id be the id of the most recent usage of the negotiation algorithm that declared X's DR policy satisfied.
2. If β is not in the ATT set then exit the algorithm.
3. Look in the ATT set to find the response, ∂, associated with the attack, β
4. Get the state of all security servers
5. For each interface, ∝, if there exists an entry (X, ∂, id, ∝) in the SERVICES list, tell one of the security servers that is allowed and able to enact response ∂ on β at ∝. Otherwise, send a **service notification (X, ∂, id, data)** message out of ∝ where data is the description of β.

[6] Our guarantees are at the design level. There is no guarantee that the implementation is correct.

See the above attack detection algorithm to see what a controller does when it receives a **service notification (X, ∂, id, data)** message.

This algorithm causes a response to occur on the attack path because the negotiation algorithm sets up a perimeter around X where only sites X trusts are in the perimeter. The attack must occur from outside the perimeter. Our response algorithm sends out **service notification (X, ∂, id, data)** messages in a breadth first search manner. The propagation terminates after a controller initiates a response on an interface. Thus, the **service notification (X, ∂, id, data)** messages will reach the perimeter set up by the negotiation algorithm and the response will occur all around the perimeter.

Service notification (X, ∂, id, data) messages do not infinitely cycle through the intranet because the data portion contains a unique identifier for each response request by X. If a site receives a **service notification (X, ∂, id, data)** message twice it disregards the second message and does not forward it.

The response generated by this algorithm may be non-optimal. It may make more sense to only initiate a response on the attack path or to initiate a response closer to the attacker. It may even make sense to initiate the response from an untrusted site. To clear up this issue, we need to do more research in optimal responses.

Chapter 3 : Optimization Algorithms for Component Arrangements

3.1 THE GLOBAL OPTIMIZER

We now present algorithms that minimize the number of security servers used. Our optimization reduces the cost of installing and maintaining the defense system described by our model. The global optimizer (GO) allows for such optimizations.

The GO is a unique process for each intranet defense system that queries the controllers at every site to find out the topology of the intranet and the state of each security component. The GO determines which sites connect to which sites, which interfaces each security server covers, the state of the each controller, the state of each security server, and the three lists that the controllers maintain to keep track of negotiations. With this information, the GO can optimize the placement of security servers within the intranet in order to minimize the number of servers needed.

3.2 OPTIMIZING THE PLACEMENT OF SECURITY SERVERS WITHIN THE INTRANET

The most obvious method is to rearrange the security servers in an intranet such that we can remove some of them and still have the intranet configuration satisfy all site's DR policies. In this section, we analyze the time complexity involved in running an algorithm to do this optimization.

Optimization 3.2.1

As input give an intranet topology, the state of every controller in every site, and a set of security servers available for installation in the intranet. Assume the intranet has no security servers installed. Place a subset of the available security servers in the intranet to minimize the number of security servers used while satisfying each site's DR policy.

Optimization 3.2.1 is useless unless we assume that the state of sites will change infrequently. When all sites change their state far enough from their original state, we will have optimized the placement of security servers for a completely different environment. In this case the GO can perform optimizations while the defense system is running and recommend new locations for some of the security servers and that others be removed.

Unfortunately, we will prove that this optimization is at least as hard as a known NP complete problem. Thus, if P is not equal to NP then there exists no polynomial algorithm to perform this type of optimization.

Decision Problem 3.2.1 (Related to Optimization 3.2.1):

Start with an intranet topology, the state of every controller in every site, and a set, n, of security servers available for installation in the intranet. Assume that the intranet has no security servers installed. Can all sites satisfy their DR policies by installing k, $k \leq n$, of the available security servers?

Lemma 3.2.1.1: Decision Problem 3.2.1 is in NP

Proof:

In order to show that decision problem 3.2.1 is in NP, we need to show that an answer to the decision problem is verifiable in polynomial time. If the answer to a particular instance of the

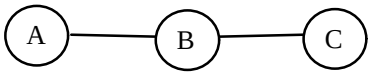
decision problem is yes, we ask for proof. The proof will be a placement of the security servers in the intranet. To verify that we used k security servers, we count them. Also, we need to make sure that this placement allows each site to satisfy its DR policy. We can run the negotiation algorithm once from each site to accretion if each site can satisfy its DR policy. The negotiation algorithm is polynomial and we are running it n times where n is the number of sites. Thus, we can verify a yes answer to the decision problem in polynomial time.

Transformation from Vertex Cover to Decision Problem 3.2.1

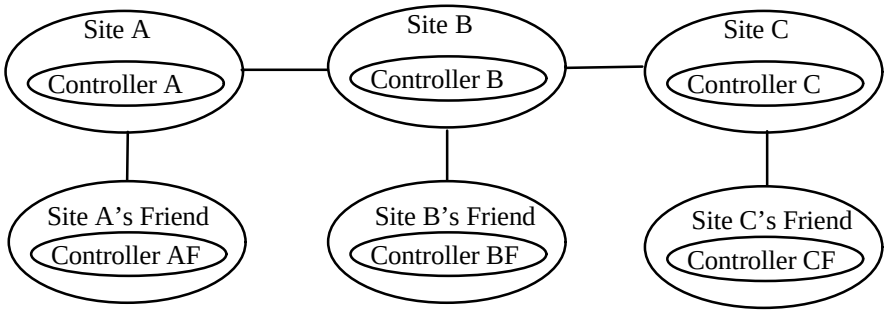
Take an arbitrary vertex cover problem[7] [16]. Let every vertex in the undirected graph of the vertex cover problem represent the controller of a site. The edges out of each vertex represent the interfaces out of the controller's site. For each controller the set ATT = {(∂,empty)}. The set TRUST for each controller is the controller's neighbors that also includes a special site called the controller's friend. Every controller, Y, has a special "friend" site. Y trusts its special friend but no other controller trusts Y's friend. The controllers at friend sites do not have any attacks or responses in their ATT set. For each vertex in the vertex cover problem, there exists a security server that is available to protect the intranet in the decision problem. Each of these security servers detects ∂ through all interfaces into the site.

[7] We define a vertex cover as follows. Take an undirected graph, G, with a set of edges, E, and a set of vertices, V. A vertex cover, V', is a subset of V such that any edge in G is adjacent to at least one vertex in V'. The vertex cover problem is to decide whether or not an undirected graph has a vertex cover of size k.

A Vertex Cover Problem:
Does there exist a vertex cover of size 2:



Decision Problem 3.2.1 :
Can all sites be made secure using 2 security servers that detect ∂ through all interfaces?



Site A:
ATT = {(∂,empty)}
TRUST = {B,AF}

Site B:
ATT = {(∂,empty)}
TRUST = {A,C,BF}

Site C:
ATT = {(∂,empty)}
TRUST = {B,CF}

Site A's Friend:
ATT = {}
TRUST = {}

Site B's Friend:
ATT = {}
TRUST = {}

Site C's Friend:
ATT = {}
TRUST = {}

Figure 3.2.1 Example Transformation for the NP complete proof of decision problem 3.2.1

Lemma 3.2.1.2 : A vertex cover of size k exists for an instance of a vertex cover problem iff all sites in the related decision problem as defined by the transformation can satisfy their DR policy using k security servers.

Proof:

First we show that if a vertex cover of size k exists, then all sites in decision problem 3.2.1 can satisfy their DR policy using k security servers. Assume that we have a vertex cover of size k. Apply the transformation to get an instance of the decision problem. For each vertex chosen to be in the vertex cover of size k, give the corresponding controller in the decision problem a security server. We claim that this will be a solution to the decision problem using k security servers.

Obviously, we have used k security servers but we must prove that every site can satisfy their DR policy with this configuration. All the controllers of "special friend" sites have no elements in their ATT set and thus by default have their DR policies satisfied. Since in the vertex cover solution, each vertex is either in the vertex cover solution or else all of its neighbors are in the vertex cover solution, each non-friend site will either have a security server or all of its non-friend neighbors will have a security server. Since the security servers detect ∂ through all interfaces, each non-friend site will either detect ∂ through all interfaces or their non-friend neighbors will detect ∂ through all their interfaces. If a non-friend site has a security server then its DR policy is satisfied because ∂ will be detected through all interfaces. If the non-friend site does not have a security server the non-friend site will be protected through all of its non-friend interfaces because all of its

neighbors are trusted and will detect ∂ for it. The non-friend site has protection through its "special friend" interface because it trusts its "special friend" and the "special friend" does not have interfaces to any other site. Therefor, if a vertex cover of size k exists then all sites can satisfy their DR policies using k security servers in the related decision problem.

Now we show that if in decision problem 3.2.1 k security servers can allow all sites to satisfy their DR policies, then in the vertex cover problem a vertex cover of size k exists. Assume there exists a solution to the decision problem that uses k security servers and still every site has its DR policy satisfied. For each non-friend site in the decision problem that has a security server, include the related vertex in the vertex cover problem into the vertex cover. Then for each "special friend" site that has a security server, pick a vertex not currently in the vertex cover to be in the vertex cover.

We need to show that the vertex cover described will indeed be a valid vertex cover. Since there are n security servers available in the decision problem, where n is the number of vertices in the vertex cover problem, there will never be more security servers used than vertices in the vertex cover problem.

Now we will finish the proof arguing by contradiction. Assume that the vertex cover described was not a valid vertex cover. This means that some edge exists where neither of its incident vertices are in the vertex cover. However, then there exists an interface in the decision problem between two non-friend sites where neither site detected ∂. But since each non-friend site has a "special friend" which is not trusted by all other sites, it can never happen that two connected non-friend sites will neither have a security server detecting ∂. If this case ever did happen, the "special friends" connected to each non-friend site would be able to launch ∂ on one of the other non-friend sites without being detected even though the non-friend sites do not trust thier non-friend neighbor's friend. Thus, we have a contradiction and the vertex cover must be valid.

Theorem 3.2.1: Decision problem 3.2.1 is NP complete.
Proof:

By lemma 3.2.1.2, we can take any instance of a vertex cover problem and transform it into a special case of decision problem 3.2.1 such that if we can solve the decision problem in polynomial time then we have a polynomial solution to the NP complete vertex cover problem. Thus, decision problem 3.2.1 is NP complete.

Now that we have shown decision problem 3.2.1 to be NP complete, we need to show that optimization 3.2.1 is as least as hard as decision problem 3.2.1 in order for us to state that if P≠NP then there exists no polynomial solution to the optimization 3.2.1.

Theorem 3.2.2: The optimization 3.2.1 problem is at least as hard as decision problem 3.2.1
Proof:

Assume that we had a polynomial algorithm to perform optimization 3.2.1. By using this algorithm on a problem instance of decision problem 3.2.1, we could come up with a placement of security servers in the intranet that would use the minimal number of security servers and still allow each site to meet its DR policy. By counting this minimal number of security servers used, we could answer the decision problem either yes or no in polynomial time.

3.3 IDENTIFYING UNNECESSARY SECURITY SERVERS

Since close variants of NP complete problems often end up solvable in polynomial time, we explore variants of optimization 3.2.1. The obvious variant is to start with an existing intranet configuration and attempt to remove security servers without rearranging them. Although this variant seems much easier than optimization 3.2.1, this problem too is at least as hard as the related NP complete decision problem.

Decision Problem 3.3.1:

Starting from an intranet topology installed with security servers such that every site can satisfy its DR policy, can we remove k security servers without rearranging the other security servers such that each site can still satisfy its DR policy?

Lemma 3.3.1.1 : Decision Problem 3.3.1 is in NP

Proof:

To prove that decision problem 3.3.1 is in NP we need to be able to verify a "yes" answer in polynomial time given proof of the answer. The proof of the answer is the intranet configuration after removal of the k security servers. We count the security servers to verify the removal of k servers. To make sure that the placement of the remaining security servers allows each site to satisfy its DR policy we run the negotiation algorithm once from each site. Since the negotiation algorithm is polynomial and we run it n times where n is the number of sites, we can verify a yes answer to the decision problem in polynomial time.

Transformation from Vertex Cover to Decision Problem 3.3.1

This transformation is identical to the transformation used in proving optimization 3.2.1 NP complete except for a minor variation in the last line. See figure 3.2.1 for a picture of this transformation.

Take an arbitrary vertex cover problem. Let every vertex in the undirected graph of the vertex cover problem represent the controller of a site. The edges out of each vertex represent the interfaces out of the controller's site. The set ATT for each controller = {(∂,empty)}. The set TRUST for each controller is the controller's neighbors. Every controller, Y, has a special "friend" site. Y trusts its special friend but no other controller trusts Y's friend. The controllers at friend sites do not have any attacks or responses in their ATT set. At each non-friend site there exists a security server which detects only ∂ but will detect ∂ through all interfaces.

Lemma 3.3.1.2: A vertex cover of size k exists for an instance of a vertex cover problem iff all sites in the decision problem as defined by the transformation can satisfy their DR policies while removing |G|-k security servers where |G| is the number of vertices in the vertex cover problem.

Proof:

First we show that if a vertex cover of size k exists, then all sites in decision problem 3.3.1 can satisfy their DR policies while removing |G| - k security servers. Assume that we have a vertex cover of size k. Apply the transformation to get an instance of the decision problem. For each vertex not in the vertex cover, remove the security server at the corresponding site in the decision problem. We are claiming that the remaining intranet configuration will be a solution to the decision problem using k security servers.

We have certainly taken away |G| - k security servers but we must prove that every site can satisfy its DR policy with this configuration. Each of the "special friend" sites have no elements in

their ATT set and thus by default can satisfy their DR policies. Each of the controllers of non-friend sites will either have a security server or else all of its non-friend neighbors will have a security server. This is because in the vertex cover solution, by definition each vertex is either in the vertex cover or all of its neighbors are in the vertex cover. Since security servers detect ∂ through all interfaces, then each non-friend site will either detect ∂ through all interfaces or their non-friend neighbors will detect ∂ through all their interfaces. Each non-friend site can now satisfy their DR policies because each non-friend site either detects ∂ through all interfaces or all of its non-friend trusted neighbors detect ∂. Furthermore, each site trusts its "special friend" not to attack it with ∂ and thus each site does not need to detect ∂ through this interface. Thus, if a vertex cover of size k exists, all sites can satisfy their DR policies removing |G| - k security servers in the related decision problem.

Now we show that if the decision problem removes |G| - k security servers while all sites are able to satisfy their DR policies, then in the vertex cover problem a vertex cover of size k exists. First, assume that we have a solution to the decision problem that removes |G| - k security servers and every site can satisfy their DR policy. For each non-friend site in the decision problem that has a security server, include the related vertex in the vertex cover problem into the vertex cover.

We will show that the vertex cover described will indeed be a valid vertex cover. First, the vertex cover will have k vertices because |G| - k sites in the decision problem had their security servers removed. Next, to set up a proof by contradiction, assume that the vertex cover described is not a valid vertex cover. This means that some edge exists where neither of its incident vertices are in the vertex cover. But this means that there exists an interface in the decision problem between two non-friend sites where neither site detects ∂. However, since each non-friend site has a special friend which is not trusted by all other sites, it is always true that two connected non-friend sites have a security server detecting ∂ between them. Otherwise, the "special friends" connected to each non-friend site will be able to launch ∂ on one of their non-friend site's neighbors without being detected even though they are not be trusted. Here we have a contradiction and the vertex cover must be valid. Therefore, if the decision problem removes |G| - k security servers and still has all sites secure, then in the vertex cover problem a vertex cover of size k exists.

Theorem 3.3.1: Decision problem 3.3.1 is NP complete.

Proof:

Lemma 3.3.1.2 allows us to conclude that we can solve any vertex cover problem by transforming it into the decision problem 3.3.1 and then solving the decision problem. Thus, if we can solve the decision problem in polynomial time, we can solve the vertex cover problem in polynomial time. Decision problem 3.3.1 is NP complete.

While decision problem 3.3.1 is NP complete, it is still important to have some approximate solution to the problem. Thus, we developed a polynomial running time algorithm that will eliminate some but not all of the unnecessary security servers.

Static Removal Algorithm

The GO obtains the topology of the intranet, the placement of security components, and the state of each component. Then, for some security server, S, at a site, X, the GO simulates the policy driver at X telling S to stop performing all detections and responses for X and all other sites.

S is effectively “turned off”. The GO simulates the controller at X using the negotiation cancellation algorithm to notify effected sites. The GO simulates the effected sites using the negotiation algorithm to obtain intranet configurations that satisfy their sites DR policies. If all sites are able to still satisfy their DR policies after S is “turned off”, then S remains “turned off”. Otherwise, the GO restores the state all components were in before S was “turned off”. At this point, the GO goes back and continues this procedure from the current, not original, state for all other security servers in the intranet.

After all processing all security servers, we can remove all that are “turned off” and the intranet configuration will still satisfy each site's DR policy.

By lemma 3.3.1 the static removal algorithm can guarantee that if it is possible to remove at least one site without rearranging sites then the algorithm will remove at least one site. There exist approximation algorithms to the related vertex cover problem that produce better upper bounds than this approximation algorithm [17]. Unfortunately, an approximation algorithm for one NP complete problem does not necessarily yield an approximation algorithm for a similar NP complete problem.

In further analysis of the static removal algorithm, if one applies the static removal algorithm to an intranet configuration, the resulting intranet configuration has hit a local minimum as proven in lemma 3.3.2. Thus, consecutive uses of the static removal algorithm produce no further benefits.

While consecutive uses of the algorithm will not be beneficial, it is beneficial to run the static removal algorithm many times starting from the same intranet configuration. Use a different ordering to process the security servers each time you run the algorithm. By lemma 3.3.3 each ordering may produce a different answer and on should use the best answer out of all the trials. If one runs the static removal algorithm on all possible orderings of security servers then they will find the optimal solution but the runtime is exponential.

One other interesting feature of the algorithm is that, by lemma 3.3.4, if the algorithm removes the same number of security servers by running with different orderings, the intranet configuration given at the output is not unique.

Lemma 3.3.1: Given an initial intranet configuration Y as input to the static removal algorithm, if it is possible to remove at least one security server without rearranging the security servers the output configuration, Z, will have at least one less security server than Y
Proof:

Assume that it is possible to remove a security server from Y and still have each site able to satisfy its DR policy but Z has the same number of security servers as Y. The static removal algorithm turns off each security server in turn and then it runs the negotiation algorithm at all effected sites to see if they can satisfy their DR policy without the security server. We proved that the negotiation algorithm satisfies a site’s DR policy if an intranet configuration exists that can satisfy the site's DR policy. Thus, the static removal algorithm would have found the removable security server and we have a contradiction.

Lemma 3.3.2: Given an initial intranet configuration Y and given that Y was used by the algorithm to generate an intranet configuration Z with fewer security servers, using Z as input to the algorithm will give Z as the output.
Proof:

During the first run of the algorithm, the GO attempts to turn off each security server in Y. If the GO turns off a security server, it does not appear in Z. Thus, each security server in Z is a security server that the GO did not turn off.

If the GO does not turn off a security server, S, during the first run of the algorithm, some controller depends upon a service that S is providing. That controller can not find any other security servers that are able to perform the needed detection because the negotiation algorithm guarantees that it will find an intranet configuration that satisfies a site's DR policy if one exists. Also, the algorithm does not add any security servers while running or give them permission to offer more services. Thus, after the first run of the algorithm, no security servers exist that will turn off on subsequent runs of the algorithm.

Therefore, the algorithm can not turn off any security servers on the second run and the output configuration will be Z.

Lemma 3.3.3: Given as input to the algorithm an initial intranet configuration, the number of security servers removed by the algorithm can change depending upon the security server processing order.

Proof:

Assume that the input to the static removal algorithm is four sites: A, B, C, and D. A, B, and C have connections to D and to no other site. The ATT set of sites A, B, and C contain (∂,empty). The ATT set of site D is empty. A security server that detects ∂ on all interfaces exists at every site. A, B, and C each trust only D.

Definition of Security for Site A:
ATT = { a }
TRUST = {D}

Definition of Security for Site B:
ATT = { a }
TRUST = {D}

Definition of Security for Site C:
ATT = { a }
TRUST = {D}

Definition of Security for Site D:
ATT = {}
TRUST = {A, B, C}

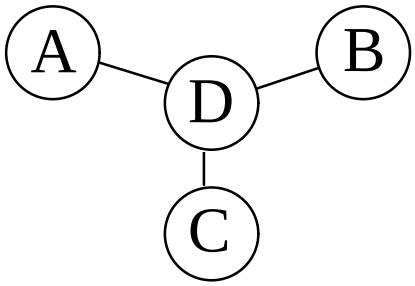


Figure 3.3.2 Example configuration for theorem 3.3.3

If the static removal algorithm processes the security server at D first, then the algorithm removes the security server at D. But then the security servers at A, B, and C are all necessary because A, B, and C are all concerned about being attacked by ∂ and they don't trust each other. Thus, if the algorithm processes the security server at D first, then the algorithm will remove only one security server.

If the security server at A is processed first, then it will be removed because trusted site D is protecting A against ∂ from one of the other untrusted sites B or C. At this point, if the algorithm processes the security server at D, it can not remove the security server at D because doing so makes A not secure. Then, the security servers at B and C are processed and removed because B and C can rely on D to detect ∂ for them. Thus, if the algorithm processes the security server at A first then the algorithm removes three security servers.

Lemma 3.3.4: Given as input to the algorithm an initial intranet configuration, the security servers removed by the algorithm can change even though it removes the same number depending upon the order in which the algorithm processes the security servers.

Proof:

Assume that the input to the static removal algorithm is four sites: A, B, C, and D. A is connected to B. B is connected to C. And C is connected to D. The ATT set of site A contains just (∂,empty). The ATT sets of sites B, C, and D are empty. No security servers exist at A and D but B and C each has a security server which can detect ∂ on all interfaces for any site. A's TRUST set is equal to {B,C}.

Definition of Security for Site A:
ATT = {(∂,empty)}
TRUST = {B,C}

Definition of Security for Site C:
ATT = {}
TRUST = {}

Definition of Security for Site B:
ATT = {}
TRUST = {}

Definition of Security for Site D:
ATT = {}
TRUST = {}

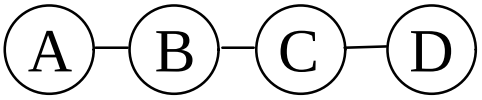


Figure 3.3.3 Example configuration for theorem 3.3.4

The static removal algorithm either processes the security server at B first or the security server at C first. Whichever security server the algorithm process's first is the one removed and the algorithm will leave the other in order to protect A from D.

3.4 IDENTIFYING TRUST CLUSTERS AND TRUST ISLANDS

While automated optimization algorithms are easy to use, we also consider algorithms that help humans identify areas in the intranet where optimizations could occur. Such algorithms can identify structures within the intranet that can lead humans to perform optimizations.

Two useful structures become readily apparent: trust clusters and trust islands. Trust clusters are sets of sites that all trust each other but can trust some other sites. Trust islands are sets of sites that all trust each other and no other sites.

These two groupings are useful to identify in an intranet because the sites in a grouping can all work together for a common defense. This common defense involves creating perimeters around adjacent sites in the set allowing fewer security servers to be needed.

Our goal is to have the GO in polynomial time identify both kinds of groups: trust clusters and trust islands. However, only one of these goals achievable because identifying trust clusters is NP complete while identifying trust islands is a simple polynomial algorithm.

Trust Cluster Decision Problem:

Given an intranet with a defense system as defined in our model, does there exist a trust cluster of size k?

Transformation: Take any instance of the clique problem[8] and let each vertex represent a site. Let every edge, E, in the clique problem that connects two vertices, A and B, represent the fact that A trusts B and B trusts A in the trust cluster problem. If we are looking for a clique of size k then we are looking for a trust cluster of size k.

Lemma 3.4.1: There exists a trust cluster of size k iff there exists a clique of size k.
Proof:

Assume there exists a trust cluster of size k. Thus, there exist k sites where each of the sites in the group trusts all of the other sites in the group. That must mean that in the related clique problem, each of the associated k vertices has a connection to all the others and thus we have a clique of size k.

Next, assume that we have a clique of size k. Thus, we have k vertices where each vertex in the group has an edge to every other vertex in the group. That means that in the related trust cluster decision problem, each of the k associated sites all trust each other and thus we have a trust cluster of size k.

Theorem 3.4.1: The decision problem of finding a trust cluster of size k within an intranet is NP complete
Proof:

This problem is NP because given a trust cluster of size k it is easy to verify that those k sites do indeed all trust each other. Now assume that we have an instance of the clique decision problem. We can transform the problem into an instance of the trust cluster decision problem using the above transformation. Then by lemma 3.4.1, the answer to the trust cluster problem is identical to the answer to the clique decision problem. Thus, if we have a polynomial solution to the trust cluster decision problem then we have a polynomial solution to the clique decision problem. However, this means that we have a polynomial algorithm to an NP complete problem. Therefore, the trust cluster decision problem must be NP complete.

Theorem 3.4.2: There exists a polynomial algorithm to find all trust islands in polynomial time.
Proof:

We prove this theorem by giving a polynomial time algorithm. Let the GO create a graph G where there exists a vertex in G for every site in the intranet. Create a directed edge in G from a vertex A to a vertex B if in the intranet site A trusts site B. For every maximal clique in G where no vertex in the clique has an edge to any vertex outside of the clique, there will be a maximal trust island in the intranet.

Create a copy of G called G'. Use a polynomial algorithm to divide G' up into its maximal strongly connected components[9] [18]. Any trust island must be a subset of a strongly connected component because for every vertex in the trust cluster there must be a directed edge to every other vertex. For each maximal strongly connected component, either the whole component is a trust island or none of the vertices are part of a trust island because in trust islands, the sites will all

[8] The known NP complete clique decision problem asks, "On an undirected graph, does there exist k vertices where each vertex in the group is connected to every other vertex in the group?".

[9] A strongly connected component of a directed graph, G, is a maximal set of vertices in G such that it is possible to traverse from each vertex in the set to every other vertex.

trust each other and no one else. Thus, only if a maximally connected component is a complete digraph will the component be a trust island.

We must test each component in G' to see if all vertices trust each other. If this condition does not hold for each vertex, the component is not a trust island. If the condition does hold for each vertex then we need to make sure that each vertex in the component in G does not have an edge to any vertex in G outside of the component. If this condition holds for each vertex in the component then the component is a maximal trust island.

Chapter 4: Miscellaneous Topics

4.1 THE PROBLEM OF TRANSITIVITY OF TRUST

In many security models the transitive properties often pose large security problems and this model is no exception [19] [20]. In our case, the transitivity problem lies with the giving of trust. Theoretically of course, no problem exists because if a site B trusts a site A then A will not attack B. And in the real world, if sites A and B were the only sites in the intranet and it was known that A would not benefit by attacking B, then B could safely trust A. However, what if a site C trusts B. A has the motivation to break into B in order to then break into C. The model will not detect this two stage attack since theoretically site A would not attack site B and site B would not attack site C.

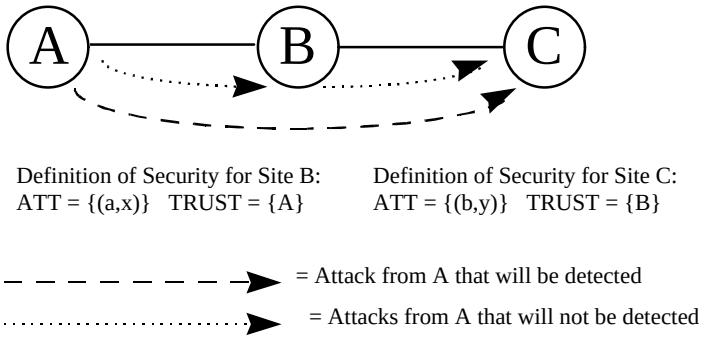


Figure 4.1.1 Example of a transitivity of trust problem

The method taken by the papers cited above to deal with transitivity problems in security models is to detect the conditions where the transitive problem could occur and then to make changes so that the transitive condition is not present. Taking this approach with our model, we can easily detect and prevent transitive conditions.

Algorithm to detect if a site X can be attacked undetected via a transitivity of trust problem

The GO obtains the current state and configuration of the intranet. For each site in the intranet the GO creates a vertex. Label blue the vertices associated with sites the controller at X trusts. Label all other sites red. If site A trusts a site B, place a directed edge from vertex A to vertex B.

To detect if a transitivity of trust attack can compromise X we start at X and perform a depth first search to attempting to reach a red or untrusted vertex using the trust relationships.

To prevent a transitivity of trust problems we define a finer granularity of trust. Instead of having a policy driver define who the site trusts, the policy driver defines who the site trusts and then who the site specifically does not trust. A site, X, does not trust sites that are not in its TRUST set but as long as these sites are not in the NOT_TRUST set then they are able to launch transitivity of trust attacks. This allows a site's policy driver to choose to not allow any transitive attack situations or to only allow them from a select set of sites. One finds violations in this variant trust mechanism using the same algorithm given above except that the red vertices are only those vertices that correspond to sites specifically not trusted by X.

The negotiation algorithm as written does not detect transitivity of trust problems but it is easy to add algorithms to detect transitivity of trust problems. One should change the negotiation algorithm so that sites can forward service request messages, when needed, to sites that are not in the TRUST set or in the NOT_TRUST set. When doing this the forwarding site should send the originator of the negotiation a message stating that it is vulnerable to transitivity of trust attacks. That change would allow sites using the finer granularity of trust to be able to use the existing negotiation algorithm. Another option to changing the negotiation algorithm is to have the GO inform sites when a transitivity violation has occurred. Upon notification of a problem, the policy driver can weaken or change DR policy. The controller then runs the negotiation algorithm to satisfy the new policy and the GO tells the controller if a transitivity problem exists with the new DR policy.

4.2 HOW CHANGING A SITE'S POLICY AFFECTS THE MODEL

Earlier we assumed that a site's policy driver would not change the site's state faster than the controller of the site could set up a new negotiation. However, we should explore what happens when a policy driver changes their site's states very often and/or drastically. In this case, a proper implementation of the model enables smooth functioning of the model under both situations.

Assume that a policy driver changes the state of a site very often. The controller at the site may have to renegotiate an intranet configuration in which the site meets its new DR policy. This does not cause problems as long as the policy driver can accept the delay it takes to set up the new negotiation. The security gained under the previous negotiation protects the site until the completion of the negotiation for the new DR policy.

What happens when changing the state at site Y causes a site X to no longer meet its DR policy? This happens when X is relying on Y for a service and the policy driver at Y prohibits Y from offering the service to other sites. In this case, X no longer meets its DR policy. X's DR policy remains unsatisfied until the negotiation algorithm alerts X of the situation and until X can renegotiate an intranet configuration in which it meets its DR policy.

The solution for this lapse in security for sites is a simple implementation modification. When a policy driver prohibits a security server from offering a certain service, the security server should delay execution of this order. The delay should be the maximum time it would take to notify the effected sites plus the time it would take the effected sites to renegotiate a new intranet configuration. This way, no sudden and unnecessary lapses in the security of sites can occur unless the negotiation algorithm is unable to find an intranet configuration in which the site can meet its DR policy.

While it appears that changing a site's state often in small ways does not cause any problems, what happens when a site changes its state drastically. Assume in this case that the policy driver changes all the sets of every security server and the controller. This also should not cause any problems except that it becomes more likely that the negotiation algorithm will be unable to find an intranet configuration that satisfies the site's DR policy.

When a site is not able to satisfy its DR policy the policy driver can try weakening the DR policy and rerunning the negotiation algorithm. If this is not possible then out-of-band (human initiated) changes can ensure that a site receives the services that it needs. In this case it may be necessary to install more security servers, enable more services at existing security servers, or persuade other sites to use policy drivers that will be more helpful in offering services to others.

While we have shown that the model handles both frequent and drastic changes, in practice we envision an intranet where the state of sites varies seldom. In this case the GO can perform global optimizations that remain useful for long periods of time before the intranet radically changes its state.

Chapter 5: Conclusions and Future Research

5.1 CONCLUSIONS

We have taken signature based IDRSs and used them to design an intranet defense system. Each site has its own detection and response policy. The negotiation algorithm automatically satisfies each site's detection and response policy. And the GO helps to minimize the number of IDRSs needed. We accomplished this by describing a distributed security application that uses IDRSs as building blocks and then by proving certain properties about our design. We now review the features of the security application that apply to solving our problem.

The security application allows each site to have its own detection and response policy. The security application contains tools that, automatically or with human guidance, install IDRSs in an intranet such that the intranet configuration satisfies each site's DR policy. These algorithms attempt to minimize the number of IDRSs used but do not find optimal polynomial solutions because theorems 3.2.1, 3.2.2, and 3.3.1 prove this problem to be NP complete.

After IDRS installation, the negotiation algorithm allows each site to configure the IDRSs such that the site meets its DR policy. Theorem 2.7.1 and 2.7.2 prove that the negotiation algorithm finds a configuration of the IDRSs such that the site satisfies its DR policy iff such a configuration exists. If a site changes its DR policy the negotiation algorithm configures the intranet to satisfy the new DR policy.

Sites are able to add new IDRSs, remove IDRSs, and change the abilities of their IDRSs while the system is running. If a change at a site, Y, causes a site, X, to fail to meet its policy then the service cancellation algorithm run by Y informs X that the intranet configuration no longer satisfies its DR policy.

A site knows at all time whether or not it meets its DR policy. The negotiation algorithm tells sites whether or not the intranet configuration satisfies their DR policy. The service cancellation algorithm informs a site when the intranet configuration no longer satisfies the site's DR policy.

This security framework and set of algorithms provides IDRSs with capabilities that they lack: the ability to allow each site its own policy, the ability to automatically configure the security systems such that a site can meet its policy, and the ability to minimize the number of security components needed.

5.2 OPEN RESEARCH AREAS

Given that this is a very new area of research, there exist many avenues to explore.

Choosing Optimal Responses to Attacks

We assume that for every attack, the policy driver specifies what response should take place but we never indicate how the policy driver chooses an appropriate response for each attack. While we scripted this choice, there may exist methods for a system to automatically choose a response given the severity of the attack and the state of the intranet.

Choosing Optimal Placement of Responses to Attacks

Our security model guarantees that a response will take place on any path from a site, X, to any sites that X does not trust. However, many responses will hurt the legitimate traffic and it may

not be necessary to response everywhere on the defense perimeter. How can we optimally choose the placement of responses in an intranet?

Detecting and Responding to Distributed Attacks with Low Site Signatures

Our model assumes that security servers can detect attacks by analyzing the network traffic flowing through an interface or at the host level. However, many distributed probing attacks or worms have such a small signature at a network interface that they are undetectable. These attacks require a distributed detection system that collects and aggregates data from many sites. The UC Davis GRIDS system is one such system. However, none of these systems have addressed where to respond to such attacks [8].

Dealing with Imprecise Intrusion Detection

Our model assumes that security servers can detect attacks with one hundred percent accuracy. While this is convenient for designing formal models, in practice it is simply not true. Can one extend the policy template to include the certainty an IDRS has that what it detects is an attack?

Handling Responses that Occur too Late

Our model assumes that responses to attacks occur fast enough that they have the desired effect on the attack. However, what if the taken response is too late What should a controller do in this situation? How can a controller detect that a taken response was too late?

BIBLIOGRAPHY

[1] : Real Secure by ISS, http://www.iss.net

[2] : Intruder Alert by Axent, http://www.axent.com

[3] : Net Ranger, http://www.wheelgroup.com/netrangr/1netrang.html

[4] : Stalker, http://www.haystack.com/newsfr.htm

[5] : Network Radar DARPA contract numbers. Phase 1 : F30602-97-C-0299 Phase 2 : F30602-97-C-0300

[6] : Phillip A. Porras, Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances, 20th NISSC, October 9, 1997

[7] : The Master Intrusion Detection System (MIDS), DARPA contract number: JG8989

[8] : S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, D. Zerkle. GRIDS - A Graph-Based Intrusion Detection System for Large Networks, The 19th National Information Systems Security Conference.

[9] : Gene H. Kim, Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker, 2nd ACM Conference on Computer and Communications Security, 1994

[10] : Computer Oracle and Password System (COPS)
http://www.fish.com/security/cops/overview.html

[11] : SPI-NET, http://ciac.llnl.gov/cstc/spi/spinet.html

[12] : DoSTracker by MCI, http://www.security.mci.net/dostracker

[13] : Next-Generation Intrusion Detection Expert System (NIDES),
http://www.csl.sri.com/nides/index5.html

[14] : The Boeing Intruder Isolation and Detection Protocol, DARPA contract number: JG8989

[15] : An Insecurity Flow Model, Ira S. Moskowitz
Proceedings New Security Paradigms Workshop 1997, Cumbria, UK; ACM Press

[16] : Introduction to Algorithms, Thomas Cormen, Charles Leiserson, Ronald Rivest, p949, copyright 1990, MIT Press

[17] : ibid. p966-968

[18] : ibid. p488

[19] : The Cascade Vulnerability Problem, J.D. Horton, pp 110-116
Security and Privacy, Oakland Conference, IEEE 1993

[20] : The Cascading Problem for Interconnected Networks, Jonathan K. Millen, pp 269-274
4th Annual Computer Security Applications Conference, IEEE 1988

[21] : StackGuard: Automatic Detection and Prevention of Buffer-Overflow Attacks, Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathon Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, DARPA contracts F30602-96-1-0331 and F30602-96-1-0302

Appendix A: Attack Descriptions

The Pepsi Attack

The pepsi attack is where an attacker wishes to cause two hosts or routers to bombard each other with useless packets. The technique is to send a UDP packet to host 1 that has host 2 as the source IP address. The destination port is the character generator port (19) and the source port is the echo port (7). Host 1 receives the packet and replies to host 2 with another packet full of characters. Host 2 receives the packet full of characters on its echo port and reflects the packet back to host 1. This causes an infinite loop until the increased network congestion causes the network to drop UDP packet. This surprisingly simply technique is capable of bringing down many networks.

The Smurf Attack

A smurf attack is an attack whereby an attacker persuades a third party network to bombard the target network with ICMP packets. The attacker sends the third party network a constant stream of ICMP ping packets destined for a broadcast address. The attacker forges the source of these ICMP ping packets to be a site in the target network. The result is that for every ICMP ping packet the attacker sends, the attack hits the target with hundreds or thousands of ICMP echo reply packets. The thousands of packets slow down the target network and often disrupt vital services.

The target network often thinks that it is being attacked by the third party and the third party sees a constant stream of packets coming from the target host. The attacker remains almost untraceable.

Honey Pots

A honey pot is not an attack but a means by which to detect an attack. A systems administrator sets up a honey pot creating a directory that would look tempting to an intruder. Administrators inform regular users that they should not enter this directory. If the intrusion detection system sees a person inspecting the honey pot directory then it sounds an alarm and reports the attacker's connection.

Remote Buffer Overflow Attacks

A remote buffer overflow attack is where a hacker accesses your computer over the network having no privilege and finds a service that will ask him for input. The hacker then gives an extremely large input and hopes that the host will not check the size of the input. The attackers design their input to be larger than the data structure that is to hold the input and the attacker can overwrite memory on the target machine. If the hacker can then trick the server into executing the overwritten memory then the hacker can execute a program on the target machine. If the server on the target host is running as root, then the attacker has compromised the target. Several automatic defenses exist to these types of attacks [21].

Password Guessing

A password guessing attack is where a hacker repeatedly tries to log into a system using different passwords and/or user names. While it is unlikely that a human could guess enough passwords to break into a system, hackers write scripts that automatically guess passwords. Since

people often use easy to guess passwords like words from the dictionary or people's names, these systems are often successful in penetrating sites.